[illegible]

2

RADC-TR-90-349
Final Technical Report
December 1990

AD-A230 621

KBSA FRAMEWORK

Honeywell, Inc.

Aaron Larson, John Kimball, Jeff Clark,Bob Schrag

DTIC
ELECTE
JAN 03 1991
D
E

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

91 1 09 014

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public including foreign nations.

RADC-TR-90-349 has been reviewed and is approved for publication.

APPROVED: Douglas A. White

DOUGLAS A. WHITE
Project Engineer

APPROVED: Raymond P. Urtz

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: BGOaks

BILLY G. OAKS
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
December 1990

3. REPORT TYPE AND DATES COVERED
Final Mar 88 to Apr 90

4. TITLE AND SUBTITLE
KBSA FRAMEWORK

5. FUNDING NUMBERS
C - F30602-86-C-0074
PE - 62702F
PR - 5581
TA - 27
WU - 14

6. AUTHOR(S)
Aaron Larson, Jeff Clark, John Kimball, Bob Schrag

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Honeywell, Inc.
Systems & Research Center
3660 Technology Drive
Minneapolis MN 55418

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-349

11. SUPPLEMENTARY NOTES
RADC Project Engineer: Douglas A. White/COES/(315)330-3564

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes research performed while developing specifications for a support environment (framework) suitable for serving as the common basis for integration and continued development of the many lifecycle facets of the Knowledge-Based Software Assistant (KBSA). The KBSA will be a system which provides significant automation in the development and lifetime support of large software systems. A history of the effort including a background for many of the design decisions and the lessons learned throughout the project are included.

14. SUBJECT TERMS
Software development, Artificial intelligence, Knowledge-Based systems

15. NUMBER OF PAGES
134

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Accession For
NTIS GRA&I ☒
DTIC TAB □
Unannounced □
Justification

By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

DTIC QUALITY INSPECTED 4

Contents

List of Figures

Part I

Project Overview and Lessons Learned

Chapter 1

Project Overview and History

The 1983 KBSA report ([1]) described the requirements for a next generation software development system. The report presupposed a collection of cooperating knowledge based experts or "facets" living in and under a common environment or "framework". It placed specific requirements on the framework, focusing on activities coordination, standard abstract data structure representations, and noted that "such standards are difficult undertakings." The activities coordinator was to be responsible for managing the interactions of individuals during the evolution of the software product, providing guidance and decision making support to the members of the software team. The "standard abstract data representations" were to provide a communication channel between the cooperating facets. The report went on to mention several other aspects of the framework, particularly version management (which was to be part of the activities coordinator), access control mechanisms, user interface support, integration technology, and data base support.

The overall KBSA project plan called for "each lifecycle phase to be incrementally formalized" as part of the activities coordinator which was to force "consistency among these activities and impose standards." *This was to be accomplished* by creating "suitable KBSA frameworks" and by the "periodic integration of the evolving individual KBSA facets into a succession of more comprehensive KBSAs."

When Honeywell started the framework contract, we believed the approach outlined in the KBSA report to be reasonable and planned to follow it. We also believed that an object oriented view of the central data model with some logical expressive capability would be a good description mechanism, and would likely be a good implementation vehicle as well. We further realized that whatever system we were to describe would have to take account of the fact that the system would be multi-user and that the data base would be, at least in some sense, distributed. We initially intended to select from among the existing KBSA facet languages as the basis for the framework language, but data rights issues precluded this. Given this situation we prototyped a distributed object management system with an integrated logic engine ([2]). The prototype was intended to give us an understanding of the restrictions likely to exist in commercial object data base systems (that were then under development) as they would relate to the KBSA environment (*e.g.*, how could we integrate them with a logic engine, how would they fit into LISP, *etc.*), and to provide a platform for doing integration testing.

In 1987 we integrated the Project Management Assistant (PMA) into the prototype framework. The purpose of the integration was to see what level of support the facet needed from the framework and to get a sufficient understanding of the facet to determine the information model that the facet used. At about this time, we held a developer's meeting to discuss integration issues. The general consensus of the meeting was that a loose integration of the facets should be possible. Initially it was our intention to use a commercial Object Oriented Data Base System (OODBS) as the foundation of the framework for integrating the facets; however, anticipated commercial products were not forthcoming — with the net effect that we were forced to continue maintaining our prototype system.

The integration of the PMA into the framework went reasonably smoothly, but it soon became apparent that we would not obtain the implied information model from the source code. Primarily, this was because the source code embodied only a small portion of the developer's knowledge of the problem. At one of the subsequent KBSA developers meetings, we proposed a cooperative development of the information model for the KBSA; however, contracting issues and immaturity of the projects made the other contractors reluctant to undertake such an endeavor.

The next facet addressed was the Requirements Assistant. As with the PMA, data rights issues delayed the delivery of the facet source code. Once the source was received, we tightly integrated a portion of it into the framework. Although this time the source was much more substantial, it was once again clear that reverse engineering the information model from the source was not feasible, primarily because much of the meaning was hidden in the frames representation (which had little discernible structure). Once again, we attempted to start a joint information modeling task with the other developers. We proposed that Honeywell develop an initial version of the information model and submit it to the other developers for review. It became clear that the developers were reluctant to commit their resources and that the model would be fairly shallow in any event, and as a result this effort was not pursued.

Throughout the contract, we worked with the developers trying to standardize on items that would either be part of the long term development of the KBSA, or which would promote interchange of software prototypes. We believed that the facets should be built on Common Lisp, and on the Common Lisp Object System (CLOS). The developers agreed that the prototypes should be based on Common Lisp, with windowing support from X windows. There was partial agreement that CLOS should be used as the base object system. It was, however, agreed that the X windows interface was too low-level to be practical, and Honeywell agreed to build a toolkit based on X (KUIE).

Throughout this time we were closely following the ANSI Common Lisp standardization process, with particular attention to the CLOS and Common Lisp windowing systems. The object level of CLOS has been adopted into the draft ANSI Common Lisp standard, with adoption of the Meta Object Protocol (MOP) delayed pending further experimentation by Common Lisp users and implementors. Although recently there have been a number of Common Lisp windowing systems released, no clear windowing standard has yet emerged.

In addition to using our framework prototype for integration experiments, we used it to prototype several Access Control systems and did numerous experiments with version and configuration management approaches. The developed LISP environment with configuration management support was heavily used during the KUIE project development.

Chapter 2

Lessons Learned

The following sections summarize the lessons that we learned during the course of the framework contract that have not been described in our other reports, and make suggestions for future directions.

2.1 Persistent Object Store

At the outset of the KBSA framework project, there were several commercial vendors of Object Oriented Data Base Systems (OODBS) expecting to have product releases "within a year". After approximately that time period, they were out of business, unable to deliver a marketable system. Recently there has been a resurgence of OODBS vendors, and we believe that market pressures will result in the development of usable OODBS within a time frame acceptable to the KBSA program (1–2 years). What is less certain is whether or not the OODBS will support the capabilities needed by the KBSA contractors. Honeywell, at the ıequest of the KBSA Technology Transfer consortium, did a survey of the KBSA developers asking what capabilities they needed/expected in an OODB. We only received one response (from USC/ISI), but based on it, our own understanding of the problem, and what we believe the other developers expect, there is likely to be some mismatch in the expectations of the developers and the capabilities of commercial OODBS. The primary issue is that of consistency constraints. The developers expect a fairly powerful, probably first order logic, constraint mechanism. Existing OODBS provide a much simpler constraint mechanism, usually a fixed set of predicates over the "slots" of stored objects. Optimizing very general constraints is quite difficult[1] and it is unlikely that commercial vendors are going to address this issue in the near future.

Another deficiency of existing OODBS is the general lack of a capability to store and use "methods" (*i.e.*, code or behaviors) in the data base. Without this capability, the constraint mechanism is limited to only refer to the "slots", or representation, of the stored objects, rather than the method based abstractions on the slots. This is a clear violation of encapsulation, one of the primary benefits of object oriented programming. Furthermore, if methods are not part of the OODB, then maintaining consistency between the stored data and the programs that manipulate it becomes a configuration management problem. On the other hand, making the OODB store and run methods makes the OODB vendor create an execution environment sufficiently general to model the control primitives the users expect (*e.g.*, should it have multi methods? multiple inheritance?, *etc.*). Furthermore, optimization of method invocation has proven to be difficult when dynamic user specialization is permitted. Additionally, transactions and configuration management issues are likely to make it even more difficult. Making all this work in a distributed multi user environment will require significant additional time and effort.

These questions of course raise the issue of how tightly should the OODB be integrated with the languages and tools that will be manipulating it? If a tight integration is chosen, the resulting system will be a huge monolithic environment; if loose integration is chosen, then maintaining data base consistency will be very difficult.

Further experimentation with OODBS supporting software development environments will undoubtedly sort out some of these issues. In the mean time, the approach taken by the various KBSA developers (*i.e.*, using a single user virtual memory scheme with wholesale load and store capability) is likely to be acceptable for experimentation of *single user* programming environments. The issues involved in coordinating multi-user projects as they evolve is partially described in the configuration management section of this report, but more work is necessary.

A centralized OODB with remote client access should be sufficient for near term research and experimentation with configuration management and individual facets. For full scale product development, a truly distributed OODB will almost certainly be necessary.

[1] It is hard to determine the domain of the characteristic function for the set of objects which could change the validity of a constraint. At least one system we have seen ([3]) dynamically computes the dependents of a constraint based on reaching definitions, but this has a fairly high overhead.

2.2 Configuration Management

Configuration Management is the management over time of systems as they evolve.

The Configuration Management (CM) section of this report details our current belief of the requirements and approach that should be taken to support configuration management in the KBSA. We believe that CM will play an important part in the eventual KBSA, having impact on a large number of issues; relationships to OODBS, activities coordination, module interconnection, and reuse. The following paragraphs highlight the experiences we had related to CM during the development of the framework prototype.

During the development of the framework prototype, we went through many revisions of not only our software but of other prototype software. We believe this to be indicative of prototype development in general, since during prototyping you are normally assembling existing systems, usually experimental themselves, to get some particular effect. One of the reasons experimental components are used is because they typically embody the latest (best) understanding of the systems you are assembling. The framework prototype, at one point or another during its lifetime, ran on three different operating system/LISP implementations (VAXLISP on VMS, Symbolics, and Allegro CL on Unix) each through several major upgrades. The framework uses as components the LogLisp Programming System [4], and PCL (a portable CLOS implementation), various miscellaneous tools (browsers, communication packages, *etc.*). In addition, KUIE is based on the Common Lisp interface to X windows (CLX), and Texas Instruments' Common Lisp User-interface Environment (CLUE). Each of these components went through numerous versions (we didn't keep track of the exact number, but a good estimate would be 6 each), and since most of the systems were themselves under development, many of the versions were incompatible.[2]

A typical statement made during development would go something like "We've just found a serious bug in version A of package X, if we could upgrade to version B of X, then the problem would be solved, *except* version B of X requires that we have version E of package Y, which we can't use because..." Unfortunately most existing version management systems are incapable of describing the sort of compatibility information necessary to manage collections of software in such an environment. It is our belief that a successful KBSA must incorporate such information in a way that can be

[2] As an interesting aside, it should be noted that during this time we were also modifying our configuration management software, which brings up the question of how to do version management of your configuration management system, or equivalently "How do you change the way you change?" This sort of meta level question is typical of object oriented systems, and hints that the problem has another level of interpretation. We did not address this issue.

reasoned about to aid in the assembly of large software systems. We have made a start at describing such a system, but further work is required. The Configuration Management section of this report explains this problem in more detail and lists several other open issues. It is clear that getting real world data to validate a CM system is going to be expensive since it means selecting a software system, changing its configuration management policy, then watching it as it changes.

In an attempt to address some of these issues, we developed a configuration management system for our LISP development. The resulting system does manage to capture some of the intermodule dependencies, particularly as they relate to compilation. Our approach makes heavy use of the UNIX file system, and the fact that we are dealing with a fairly course granularity. The prototype (we call it the `cle:defsystem`) has been used internally on many projects and several other researchers (outside of Honeywell) have acquired it. The people here doing Ada development have adopted several of the principles and are currently using it for Ada software development.

2.3 Access Control

We prototyped an access control system based on a Common Apse Interface Set-like role/capability model, (which would be considered discretionary access control by [5]), which was described in [2]. Our belief was that each newly created object would get associated with it a list of "roles" which defined the access that various users would have to the object. We believed that this capability would be useful in Configuration Management (*e.g.*, freezing a version), activities coordination (*i.e.*, enforcing policy), and discretionary access has proven useful to prevent unanticipated catastrophe (*e.g.*, issuing the "delete all files" command when you aren't where you thought you were).

The access control policy that we described has a number of faults. First, it is usually unclear what the roles for a newly created object should be, normally depending not only on the user, but the task the user was doing when the object was created and what kind of an object it was. This implies that a more general mechanism, perhaps based on some underlying more general constraint mechanism, is needed. Similarly access to an object is necessary, but not sufficient, to describe activities coordination policy. This leads us to believe that an object's relationships to other domain objects is a more appropriate vehicle for describing access and policy than are the object's relationships to some artificial "roles" or capabilities. As for the needs of the configuration management system, our current belief is that a simple "read only" mode is all that is necessary, probably with "copy on write" as the default behavior for "access violations". Since this does not depend on any other state of the system, a one bit flag associated with each object is all that is necessary.

2.4 User Interfaces

Current folk wisdom is that about a third of the cost of a system is in user interface development, and it tends to be the part of the system which is least portable. With the advent of the X windowing system ([6]), the portability issues are somewhat alleviated, but building user interfaces still remains an expensive and time consuming task. The situation is improving due to the development of various window system toolkit implementations of "widgets" (graphical objects, usually windows with a particular behavior, *e.g.,* a scroll bar), and some tools for constructing user interface layouts (usually some sort of draw-like program).

Typically, a "widget" with some particular behavior or appearance is associated with a specific application object. Since the widget is represented explicitly as an object (as opposed to implicitly by a call to a draw function), both application programs and widget management systems can define behaviors for it. For example, the widget manager can automatically maintain its screen appearance and reason about its relationships to other widgets (things like stacking order, position, and visibility). The widget can also manage protocol interactions between the user and the application object (*e.g.,* "when the user presses the mouse, call the following function with the following arguments..."). This sort of widget capability is available in several of the X toolkits and some of the Common Lisp based window systems ([7, 8, 9]).

Although the widget capability described above does reduce some of the burden of user interface developers, it still requires a substantial amount of programming to correctly interpret a multi-event sequence caused by a user manipulating a widget (*e.g.,* moving a widget by having it follow the mouse may involve handling a half a dozen different kind of events). Even after the mechanics of the operation have been programmed, application programs must then interpret the intended user action based on the resulting spatial relationships of the widgets (*e.g.,* Did the user release the mouse button when the dragged widget was over the garbage can? Was it to the right or left of the XYZ widget?, *etc.*). Notice once again that the information involved is implicitly represented in the control flow of the program.

The goal of KUIE is to provide a toolkit for LISP programmers which permits them to declaratively (explicitly) state as much of their intentions for the appearance and behavior of the user interface as is practical. KUIE is structured in three layers, levels 1-3. KUIE level 1 provides a collection of graphic widgets (*e.g.,* boxes, circles, lines, polygons, connections, *etc.*) and composites (widgets which hold other widgets). KUIE level 1 explicitly represents the appearance of an object (border color, size, shape, *etc.*), its parent/child relationships to other graphics, its "stacking order" (*i.e.,* "behind", or "in front of"), and its position. Changes to any of the defined attributes are automatically realized in the disp'ay. The primary goal of the second layer of KUIE is to permit the programmer to declaratively state the spatial relationships between objects so that KUIE can manipulate and reason about them (and do automatic placement). For example, stating explicitly that the "X" widget should be "to the right" of the "Y" widget permits KUIE to reason about the situation where the user attempts to move the "X" widget "to the *left*" of the "Y" widget. By introducing graphical widgets, a substantial productivity gain has been realized because the runtime system can automatically deal with issues that previously had to be handled by application programmers (*e.g.*, maintaining the screen appearance). We believe that by making the spatial relationships explicit, we will be able to realize a similar productivity increase since the runtime system will be able to recognize changes to higher level spatial relationships, thus freeing the application programmer from having to do so.

Chapter 3 of the KUIE Reference Manual ([7]) describes the automated layout portion of KUIE which manages the satisfaction and manipulation of spatial constraints for graphics. We believe that when this system is complete, creating interactive user interfaces will require substantially less detailed specification than is currently necessary, resulting in a substantial amount of time and money savings for interface construction.

On a somewhat unrelated note, we have joined the body of researchers which have found that developing a visually pleasing interface requires the aid of a graphic designer. When we received some color workstations, we put together a color selection program and proceeded to make color selections for our window applications. The results were less than satisfactory.

2.5 Hardware/Software Issues

Early in the framework contract, we realized that a system which was based on proprietary hardware and software (*e.g.*, Symbolics Lisp machines) would not be acceptable to the general computing industry, and it was apparent even then that vendors of special purpose hardware were going to have a hard time keeping up with the big chip manufacturers With this in mind, we decided to divorce our prototype from the LISP machine environment, and encourage the other developers to either switch themselves, or at least to realize that the eventual delivery platform was likely to be a UNIX derivative and to plan accordingly. Throughout the development of the framework prototype we used freely available software, or software which was available from multiple vendors, thus making the resulting system more accessible to other researchers.

We still strongly believe that the eventual KBSA system will have to be freely available in prototype form so that distribution to other researchers will be possible. If the prototype is successful, commercial versions will then become available from vendors (perhaps initially based on the prototype). It is also fairly clear that a UNIX derivative should be the target delivery platform. What is not as clear is when the KBSA developer prototyping should be moved to a more traditional delivery platform (away from LISP machines). Too early a move will result in higher prototyping costs, while too late a switch will result in rejection/inaccessibility by/to the rest of the software developing community (both research and commercial). Several of the LISP vendors (LUCID and Franz, at least) are working on a LISP delivery platform having higher performance and requiring less system resources than current implementations, so switching from Common Lisp may not be critical, but moving away from the proprietary hardware will be necessary.

A model currently being used successfully in the UNIX marketplace is to have a working "prototype" publicly available so that prospective customers can experiment with the system for a minimal cost (usually free), and then have a commercial version available once the customer is "hooked". This approach could work well for moving the KBSA from research to industry.

2.6 Facet Integration

Although it was believed at the outset of the framework contract the facets could be integrated in a "loose" manner, it eventually became clear that this was not likely to be successful. The model of development currently held is that the development of a system using the KBSA will be iterative, requiring the simultaneous refinement of several parts of the system at multiple levels (requirements, specification, code) as more knowledge of the system becomes available. This, coupled with the fact that most software systems (the KBSA prototypes included) never have a clear, rigorous specification of their own data, makes loose integration nearly impossible. Integration of one prototype facet with another will have to be viewed, essentially, as a new development. A problem that exacerbates this is that the current KBSA fault expectations of the other facets have not been communicated very well. As stated in Chapter 3, "Vision of the Framework" below, we believe that development of a central information model is necessary for a successful KBSA system. We believe that development of this model should start as soon as possible.

2.7 Prototyping Language Choice

The current implementation language of choice among the KBSA developers is Common Lisp. We believe this is appropriate. It provides a platform for fast program development with the typical benefits of LISP (automatic memory management, good introspective capabilities, extensible *etc.*) while still providing reasonable performance. With the advent of Common Lisp, many commercial vendors now have good CL implementations, and there are even some free ones, permitting fairly good distribution of results. The adoption of CLOS and exception handling into ANSI Common Lisp should help to make programs even more portable. The lack of a standard windowing system may be somewhat minimized by the use of CLX as an interface to X windows, but since it is so low level, windowing system support will continue to be a problem for the developers. The adoption of KUIE, CLUE, and CLIO would help, but with the recent move by the major vendors to support CLIM, the correct choice is not clear. CLIM is heavily based on the Symbolics Dynamic window system, and is primarily intended to support display and input of "objects". KUIE, CLUE, and CLIO emphasize interactive graphic and text manipulation capabilities. The systems are somewhat complementary, but it remains to be seen how much interoperability there will be in practice.

With more and more toolkits being written in C and C language derivatives, it is clear that being able to make use of them in the KBSA prototyping efforts is desirable. This requires a significantly better LISP "foreign function" capability than is currently available.

2.8 Object Oriented Systems

During the course of the KBSA framework, we gained a significant amount of experience in using and writing object oriented systems, in particular CLOS. We started using CLOS when it was still thought that a simple extension to `defstruct` would be sufficient to handle object oriented programming in Common Lisp. We learned, along with the rest of the Common Lisp community, that building object oriented systems truly does offer many of the advantages typically associated with them, namely good reuseability, extensibility (flexibility in the face of change), good structure, and abstraction. There are, however, some very sticky issues (some of which are fairly subtle) that must be understood before the benefits of object oriented programming can be realized.

One of the last things we learned and probably the most important, is that writing specifications for object oriented systems is incredibly difficult. The primary cause of this is that object oriented systems do something that few other kinds of programming systems do — they "call back out". In other words, when a call is made to a "generic function"[3] it is very likely that it will call some other generic function which the caller may have specialized (modified the behavior of). The problem is that if the circumstances under which the second generic function might be called are not very well specified, it is nearly impossible to specialize it correctly. The problem is further exacerbated if the protocol must incorporate some flexibility concerning the circumstances in which the method will be called (*e.g.*, for optimization purposes, when it is frequently desirable to avoiding calling the specialized generic function under some circumstances). This is primarily caused by the fact that specifying when a particular generic function will be called exposes part of the underlying algorithm. Determining how much of the underlying algorithmic process to expose and what parts to hide is a difficult problem; essentially it is equivalent to predicting in advance the ways in which the system will be extended in the future. Specifying this requires a flexibility/optimizability tradeoff during the design of the system, something which has not typically been done rigorously in the past.

The last object oriented system for which we have written a specification is the KUIE user interface toolkit. The textual description of KUIE is between 3 and 4 times the size of the source code and a (subjective) estimate is that only about 60% of the "ideal" specification has been captured. This fact alone makes one wonder if English is an appropriate mechanism for specifying object oriented systems. Perhaps a better way would be to have the source code stand for itself either with some stylized commenting or perhaps by annotating the source to describe what part is the (hidden) implementation and what part is the visible specification. This issue is almost certainly going to arise in future stages of the KBSA program (documentation facet?). The question to consider now is how much stylized notation is permissible in a "specification" document? Too much notation makes it difficult to get a good overall *understanding* of the problem; too little notation makes the specification too imprecise as a specification document.

On an implementation level we have found that some "simple" problems cause a substantial amount of difficulty in real systems. Two examples; first, we've found that selecting the right name for a generic function is surprisingly difficult since the name, in a small but practically meaningful way, describes your expectation of how the function will be generalized. For KUIE, we essentially resorted to sticking a prefix in front of all function names ("contact-") and then using the most generic name for

[3] In CLOS, a function whose behavior is described by a collection of methods is called a generic function.

the operation that seemed reasonable[4]. The second example is the use of slots (via `slot-value`) in the methods that specialize on a class. We have found that the use of `slot-value` is usually a needless breakdown of the rules of encapsulation, even in a specialized method, and results in more fragile code (more subject to bugs). These are just two examples of the many stylistic rules which need to be developed by the practitioners of object oriented programming before it can be successfully used by the general software development community.

[4] Other approaches include placing the class name in front of the generic function which first defines it, or using only the operation name. In practice it is difficult to read programs that use the first approach since it leads to many special cases and the second approach results in many name collisions in the Common Lisp package system.

Chapter 3

Vision of the Framework

The eventual KBSA framework will undoubtedly be based on a distributed persistent object oriented data base. For the foreseeable future it will have to be accessible by programs written in several different programming languages, otherwise the current huge mass of existing software will be inaccessible. We believe that one of the major obstacles to software development for the KBSA is determining an appropriate representation for information regarding the evolution of a system. Another is coordinating the activities of a development team. The unifying theme for these two problems is the ability to represent and reason about the change between acceptable states of a software system (*i.e.*, changes between releases). Managing and reasoning about change will be a very large piece of the input to the policy engine of the KBSA activities coordinator.

If the KBSA program is to be successful, the various facet developers must begin developing a shared vision of the KBSA system. One plausible way that this could be brought about would be to start developing the information model for the KBSA. Initially, this could be nothing more than a description of the data and operations available for each of the facets. This would quickly highlight what assumptions each facet had about the information expected from each of the other facets, and would hopefully begin to increase the bandwidth of communication between the facet developers. Its primary benefit initially would be to force discussion and provide a common description of the information model. A longer term benefit would be that it would evolve into the specification for the KBSA, capturing the intermediate decisions made by each of the facets. Historically this has been difficult. It has been argued that each of the facet developers has enough problems to contend with, and that work on a central information model would take away time and energy from the development of the facets. This is true, however if work on a central information model is not begun, there is little to ensure that the facet developers will not diverge to such an extent that a unification of the facets would become a practical impossibility.

The current standardization successes (namely the use of Common Lisp, CLOS, and X windows) does very little to unify the facets. It does permit them to (perhaps) peacefully coexist in a single system, it does not in any significant way make them interoperate.

Part II

Change and Configuration Management Model

Chapter 4

Introduction to CCM

Design objects — hardware designs, software modules, document sections — are typically highly interdependent. A document, for example, is often produced as multiple files of text and diagrams, including reused boilerplate. Similarly, a software system consists of multiple modules, reuses routines from various libraries, and relies on the services of the operating system, various daemons, *etc.* Figure 4.1 shows a hypothetical document, consisting of seven objects — four text files (*.tex) and three graphics files (*.ps). Cm.tex has three components on which it depends, intro.tex has one component, and model.tex has two. (The document has other dependencies, not shown in this diagram — dependencies on a typesetting program which accepts input of a particular format, *etc.*) Design objects are seldom monolithic, existing in isolation; rather, they exist as components of other objects, or as utilities used by other objects.

Design objects are also subject to repeated change over their lifetimes — to correct errors, to cope with changing requirements, or to add new features. We call the various states in the evolution of an object the *versions* of that object; each state (except the first) is a descendent of some other state(s), and may be an ancestor of still others.

For any given use of a particular design object (or system of design objects), some versions of that object (or system) will satisfy the need, and other versions will not. Text file "exper.tex" needs to be updated with new information; behavioral model "ALU_C9" has a particular bug which the current application exercises; network server "dump-daemon" uses a protocol different from the one required. The phrase "version N is required" — as in "X11 release 4 is required" — is ubiquitous.

These characteristics of design objects — their interdependency and tendency to change — are the motivation to find techniques and tools to manage change effec-

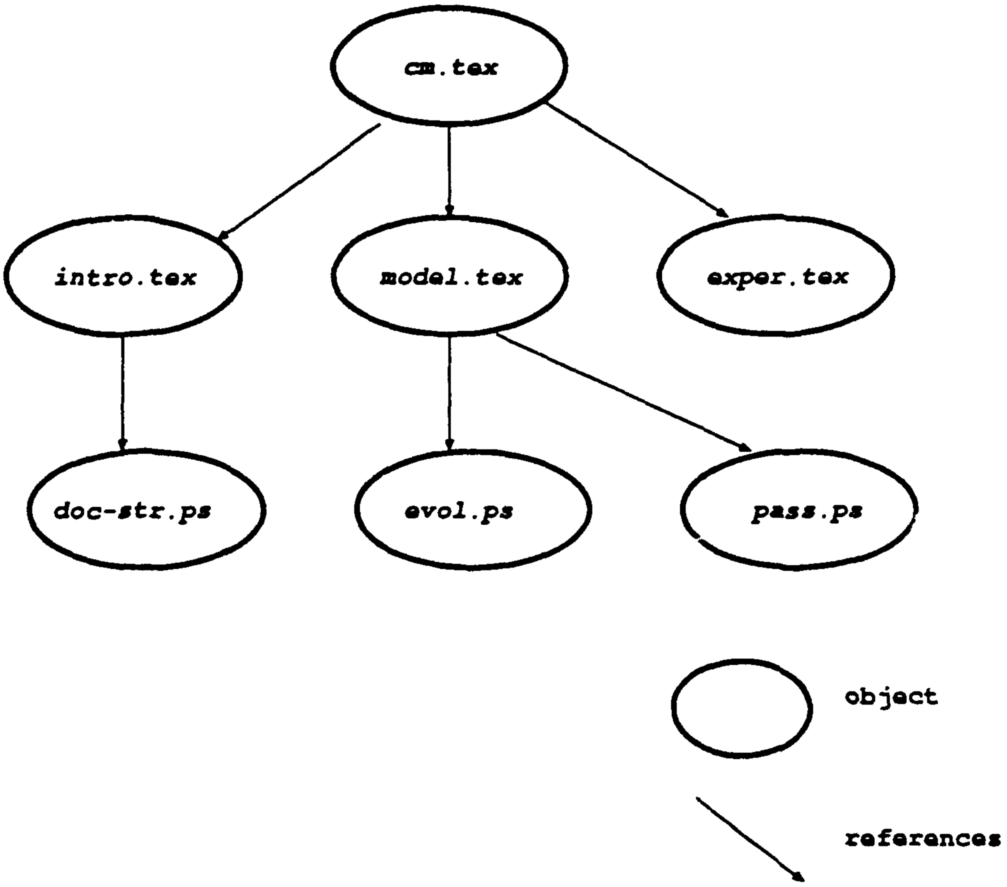

Figure 4.1: A document and its components.

tively. When object A depends on object B, and B changes, then A (and the objects that depend on A) may have problems. When an object depends on a web of other objects, and many of these objects change over time, then determining and maintaining consistency, compatibility, and equivalence becomes a problem. Whenever such a system must be changed in any way, its massive complexity is a dangerous occasion for confusion and error. The introduction and management of change, and the integrity of configurations when change has occurred, is a constant concern.

Change and configuration management (CCM) is a set of abstractions, techniques, and tools which assist in managing the evolution of systems of interdependent design objects. Over time, change occurs repeatedly as the systems of objects are modified by multiple agents. The systems of objects have class- and application-specific definitions of consistency.

We have developed a CCM model for the KBSA framework, which encapsulates our current understanding of the requirements and approach which should be taken to support CCM in the KBSA. This paper describes that model. It also describes the experiences of an in-house project which is implementing part of the model to solve CCM problems in an avionics design capture system.

4.1 Dimensions of the CCM Problem

There are a variety of projects in both academia and industry working on CCM issues. Some concepts and terminology are relatively well agreed-upon, but many others are uncommon, or are used by different groups to mean different things. Chapter 5 thus describes the dimensions of the CCM problem.

We can summarize the dimensions of the CCM problem as:

Interdependency, change, and history-bearing objects. Design objects are highly interdependent, and change over time. Multiple states of design objects must be maintained, to support recoverability and parallel evolution.

Configurations. A *configuration* is a system of design objects which work together to serve a purpose; the system of objects must be "sufficiently consistent" to satisfy that purpose. Because configurations evolve over time, a configuration may be regarded as a snapshot of one consistent state of a system of design objects. The evolution of configurations creates *version history DAGs* (directed acyclic graphs) of configurations, related by ancestor/descendent relationships.

Composing Configurations. Multiple donor configurations (*e.g.*, predecessor configurations, libraries of reusable modules) are typically composed in the process of creating or evolving a configuration. The entire donating configuration (all its objects) might be loaded into the recipient configuration; or, selected objects might be loaded into the recipient configuration; or, the donating configuration may be treated as a black box — certain of its external objects are referenced, but its other objects are hidden from the recipient configuration.

Modularization and Interfaces. Questions of modularization and interfaces — which are more typical in the programming language and software — must be considered in the design of a CCM model and system. Reusing configurations is greatly simplified if configurations have clearly-defined interfaces, and support a distinction between external and internal objects, so that configurations can be used as black boxes. When abstractly specifying a configuration, it will be important to specify what objects the configuration *needs*, and what objects it *provides*.

Design Transactions. The specification and construction of configurations is a complex and difficult task which requires the effective cooperation of multiple agents. This task can be modeled by the notion of a *design transaction* — a long-duration sequence of operations, performed by multiple agents on a shared system of objects, which starts with one configuration ("check-out") and yields another consistent configuration ("check-in", or commit). Committing the transaction involves verifying the consistency of the resulting configuration; it adds a new node — a new configuration — to the history DAG.

Equivalence and Search. A recurring theme in the CCM problem is the issue of equivalence and compatibility. Certain objects are equivalent to one another with respect to particular operations in particular circumstances; a central task of CCM is to know — to track or to determine — which objects are equivalent, with respect to particular operations (*e.g.*, link, compile) in particular environments (*e.g.*, "SunOS 4.0.3c using the DNS-based C runtime library", "Allegro LISP with PCL 12/88 loaded"). CCM can be thus cast as a search problem: locate an appropriate object (or system of objects) to satisfy the current need in the current context.

Dynamic Version Binding. When specifying a configuration, we typically want to refer to versioned objects. We may refer to a versioned object by a static reference which denotes a single definite version, fixed at specification time; or we may refer to a versioned object with a *dynamic reference* — a reference which will be resolved to a particular concrete version later, typically at configuration-construction time. Underpowered dynamic references point to a "default" version of an object, where the default changes as the object evolves. Fully general dynamic references function as search rules for locating an appropriate object, based on the current operation and environment.

Change Notification and Change Propagation. When a design object changes, objects which depend on that object may need to react to that change, to re-establish consistency; dependent objects might update themselves, update other objects, or notify a human. When an object changes, dependent objects must be notified, and consistency re-established, but the percolation of change must be controlled — every object which needs notification should be notified, but we must avoid notifying every object in the object-base at every change.

Objects with multiple representations. Design objects are frequently multi-representational. The same conceptual object is depicted by several different representations, often at different levels of abstraction. For example, an ALU hardware design may be represented by a layout object and a netlist object; a program may be represented by a spec object, a source code object, and an object-file object. Representations are related to each other by transformations. A *primary representation* is produced with human input. A *derived representation* is generated from another representation by application of a transformation (*e.g.*, the compile transformation, which derives relocatable-object from source-code); if the transformation is purely mechanical, the derived representation is a *secondary representation.* Multiple representations and the transformations between them must be managed in order to track and maintain consistency.

4.2 KBSA Framework CCM Model

Here we summarize the CCM model for the KBSA framework; Chapter 6 of this report explains the model in detail.

Our model addresses the dimensions of the CCM problem using these key ideas:

Configurations as Contexts for State. CCM is a global issue; it cannot be dealt with locally, on an object-by-object basis. In particular, it does not make sense to create a new version of an object by copying the object, because it is frequently impossible to halt copy-propagation before a large fraction of the object-base is duplicated. If objects may have multiple states, data access by (object, slot) no longer makes sense; all references to the state of an object must be made *in the context of some configuration.* Every data access must specify not just the object and slot, but also the configuration in which the access is to be performed: (object, slot, configuration). Thus, in our model, a configuration is a repository of state — it holds the state of the objects occurring in it, mapping object × slot → value.

Configurations as Deltas. A configuration need only record the *changes* in state which were made during one design transaction — a successor configuration is a delta from its predecessor. This leads to a space-efficient representation of a tree of configurations; with appropriate design, the representation can also be time-efficient (*a la* RCS [10]).

Compatibility Attributes. Besides objects' state, configurations also include *compatibility attributes*, which annotate the history DAGs with *satisfaction DAGs*; this information is used when de-referencing version cursors.

Transactions as Contexts of State. The initiation of a design transaction creates a *transaction handle* or proto-configuration. The transaction handle is a work context; it is a writable configuration — a repository for the state of a set of objects under CCM. Transaction handles are long-duration, sharable, and atomic; they may be nested to provide a hierarchy of workspaces. When the transaction is successfully committed, the state of the transaction handle becomes read-only: the transaction handle becomes a configuration, guaranteed to be a consistent system of objects, based on class- and application-specific definitions of consistency.

Configuration Schema. It is useful to be able to specify a configuration abstractly, including component references which are not resolved until configuration-construction time. A *configuration schema* specifies how to build (or recognize) a consistent configuration, and how to correctly propagate change notifications. A configuration schema specifies: the structure of the configuration (including rules for identifying objects to be fetched from other configurations); how to verify the consistency of the configuration; how to construct the configuration; and how to control the propagation of change within and between configurations.

Change Notification and Change Propagation. The change propagation problem is simplified by considering configurations and transaction handles as state repositories linked by cross-configuration references. One simplification is the ability to distinguish between cross-configuration change propagation and intra-configuration change propagation; these two typically require different strategies.

Cross-Configuration References and Dynamic References. A configuration schema may include references to objects which will be fetched from other configurations — cross-configuration references, or *version cursors*. The references may be dynamic — ie, the particular source configuration may not be chosen until configuration-construction time. A dynamic version cursor includes (a) the object signature and (b) a rule for selecting an appropriate version of that object (ie, an appropriate source configuration). Dynamic version cursors provide for flexibility in evolving configurations, and allow references to hypothetical objects which will be constructed in other transactions.

4.3 Initial Experience Using the Model for Avionics CAD

An in-house project is implementing part of the KBSA framework CCM model to solve CCM problems in a prototype avionics design capture system. Here we summarize their experience; more detail is provided at the end of Chapter 6.

In this avionics CAD domain, multiple teams of designers are evolving large and complex subsystem designs which must be periodically integrated to yield a design of the complete system. A key requirement is to have an on-line model of the system's baseline configuration, as well as options which may be instantiated to construct particular concrete configurations; *it must be possible* to determine the consequences of the choices which were made in constructing a particular concrete configuration.

It was necessary to retain the revision history of these avionics parts objects. Saving the entire database — a snapshot of the world — each time it was necessary to retain a configuration would require approximately 300 gigabytes per class of aircraft; this was deemed undesirable.

The first-cut solution was to maintain a revision history on an object-by-object basis; an object was checked-in by marking it "immutable"; any future changes would need to be made to a new version of that object — ie, to a mutable copy of the object, created by a check-out operation. But when checking-in or checking-out an object A, the objects which A references and which reference A must also be dealt with (see Section 6.1). The difficult question was how the objects which reference an object should respond to a new version of that object. Copy propagation problems arise; the objects which reference the old version of the object may need to themselves be duplicated, since a new version of the object referenced is a change to a feature of the objects which reference it. Thus it was necessary to produce not only a new version (copy) of the object to be checked-out, but also potentially of all the objects which referenced it — in the worst case, copying the entire database again.

Criteria which could be used to limit the recursive copy propagation without sacrificing correctness and consistency were not readily apparent; we contend that this will be true of most domains. The KBSA CCM model, which is conceptually clean and leads to a space-efficient representation of a tree of configurations, was therefore chosen.

An additional requirement for this avionics domain was to facilitate the work of multiple cooperating teams. The use of dynamic version cursors satisfies this need, allowing the multiple subdomains to be decoupled into multiple trees of configurations. If an object in subdomain X must reference an object in subdomain Y, the object's definition is not referenced directly; instead of a static reference, a dynamic reference is used — a query statement (dynamic version cursor) which may be "satisfied" by zero or more objects in the Y subdomain. A list of unsatisfied queries is maintained, and the list must be cleared before a consistent release can be captured; this is the "system integration" (consistency establishment) process.

This dynamic version cursor capability also supports a change notification facility which notifies a user of an object if the object has been changed such that it no longer satisfies the version cursor's selection rule.

The prototype avionics design capture system has been implemented in Common LISP, and tested with small test problems; it is now being exercised with real data (9000 objects, about 10 megabytes of avionics design data).

Chapter 5

Dimensions of the CCM Problem

There are a variety of projects in both academia and industry working on CCM issues. Some concepts and terminology are relatively well agreed-upon, but many others are uncommon, or are used by different groups to mean different things. Chapter 5 thus describes the dimensions of the CCM problem.

5.1 Interdependency and Change

> Design objects are highly interdependent, and change over time. Multiple states of design objects must be maintained, to support recoverability and parallel evolution.

Change and configuration management (CCM) is a set of abstractions, techniques, and tools which assist in managing the evolution of systems of interdependent design objects. Over time, change occurs repeatedly as the systems of objects are modified by multiple agents. The systems of objects have class- and application-specific definitions of consistency.

The CCM problem arises because design objects — hardware designs, software modules, document fragments — are interdependent and subject to change.

5.1.1 Interdependency

Design objects are *composite* and *interdependent* — composite because they have components, which are themselves design objects; interdependent because they utilize

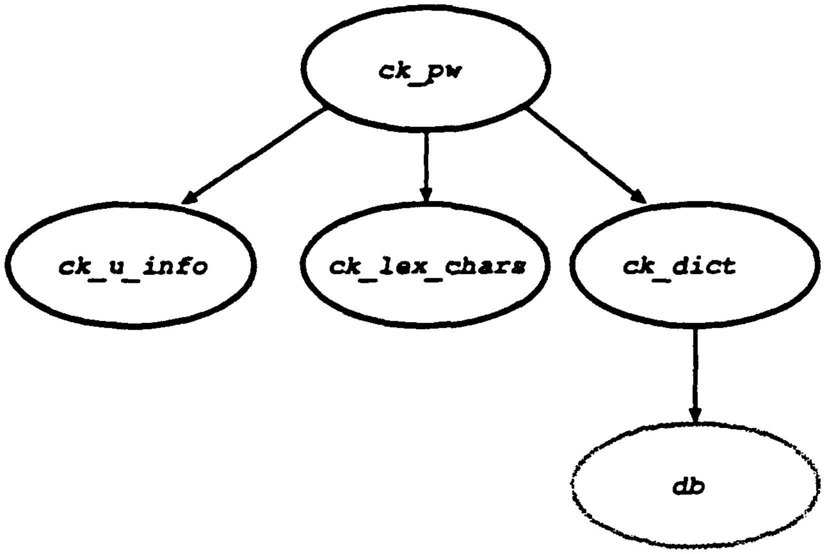

Figure 5.1: Check-password-quality program.

the services of other objects. Design objects tend to be internally complex and highly interdependent.

Design objects do not exist in isolation, but rather as *components of other objects*, or as utilities used by other objects. Figure 5.1 shows the components of an example system of design objects, a password quality-checking program, ck_pw.[1] The program references three components; one of those components, ck_dict, itself has a component, db.

5.1.2 Change

Design objects are *evolutionary*. They change over time. They are subject to revision, and the updates made to them are iterative and tentative — iterative because the objects are changed repeatedly, and tentative because it is sometimes necessary to undo a change, backtracking to a previous state.

The evolution of design objects is often *nonlinear*, but rather it branches and joins: a given state may have multiple descendents, and even multiple ancestors. Small corrections and improvements during development will produce linear (successive) evolution. But the need to provide alternate implementations or divergent functionality leads to branching. It is often necessary to maintain multiple parallel threads, all

[1] The structure and evolution of the password-checking and password-changing program examples is based very loosely on Clyde Hoover's npasswd program.

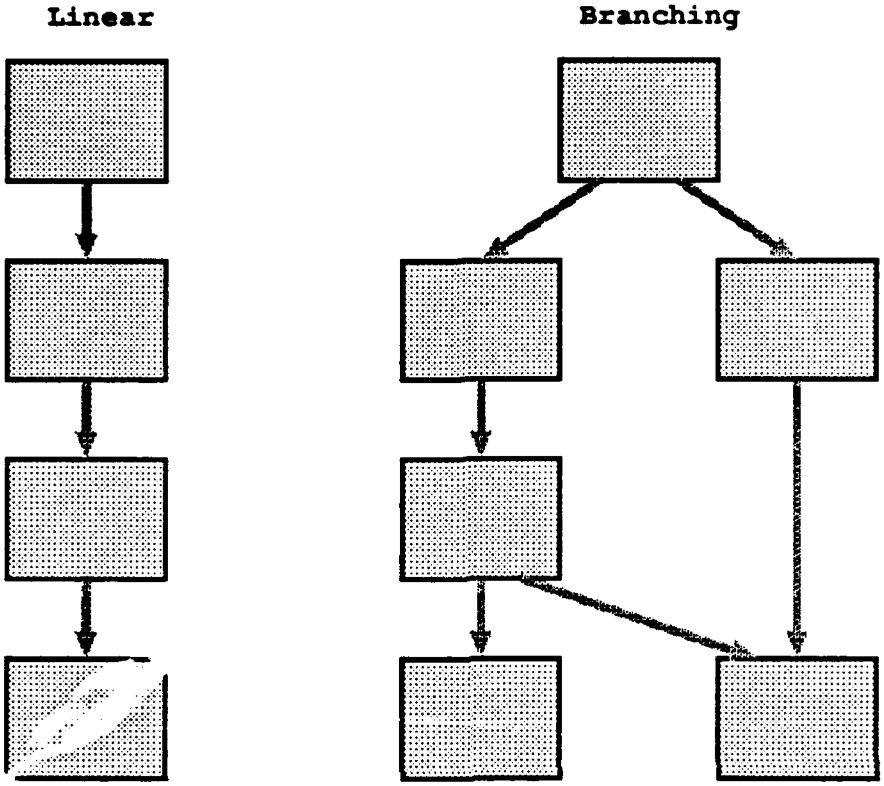

Figure 5.2: Design objects and systems of design objects are evolutionary.

evolving concurrently. For instance, a Unix-hosted tool may be at some time ported to VMS, and both threads thereafter evolve in parallel (with or without occasional joins). It is also common to find that while a designer is adding new features to an existing object for future release, other engineers are integration- and system-testing configurations containing previous versions of that object, and still other engineers are supporting various fielded releases which contain even earlier versions of the same object. We'll term the ends of these threads evolving in parallel "alternates".

Figure 5.2 shows a linear history of configurations, and a branching history of configurations which includes both a branch and a join (ie, a merge of two configurations into one). Typically, a vanilla check-in operation yields a new sequential version; a branch — the creation of an alternate — is caused by designer fiat.

Because of the needs to backtrack and to evolve multiple states in parallel, it is insufficient to record only the current state of the world: design objects must be *history-bearing*. When an object does not maintain history, updates to the object overwrite the current data; when an object is history-bearing, an update has the side effect of causing the the previous state to be saved (in some recoverable form) as historical. (In many traditional systems, an update to a history-bearing object actually causes a new object "of the same name" to come into existence.)

These characteristics of design objects — their interdependency and tendency to change — are the motivation to find techniques and tools to manage change effectively. When object A depends on object B, and B changes, then A (and the objects that depend on A) may have problems. When an object depends on a web of other objects, and many of these objects change over time, then determining and maintaining consistency, compatibility, and equivalence becomes a problem. Whenever such a system must be changed in any way, its massive complexity is a dangerous occasion for confusion and error. The introduction and management of change, and the integrity of configurations when change has occurred, is a constant concern.

To address this need, CCM techniques and tools have been created, and continue to be developed and enhanced.

Further, as object-bases become more prevalent as the foundation for design efforts, it is becoming obvious that CCM capabilities need to be part of the core services offered by those object-bases.

5.2 Configurations

> A *configuration* is a system of design objects which work together to serve a purpose; the system of objects must be "sufficiently consistent" to satisfy that purpose. Because configurations evolve over time, a configuration may be regarded as a snapshot of one consistent state of a system of design objects. The evolution of configurations creates *version history DAGs* (directed acyclic graphs) of configurations, related by ancestor/descendent relationships.

Design objects typically exist as components of other objects, or as utilities used by other objects. A consistent system of design objects which work together to meet a need is a *configuration.*

Design objects, and systems of design objects, must be history-bearing. The typical practice is to maintain history in the form of snapshots; rather than journalling every change to our set of design objects, we periodically take a snapshot of its state. As designers work on a system, the system of design objects is evolved until it reaches a state worth saving; the state of the system is then saved as a configuration (and used as a baseline for further evolution). *A configuration is thus a snapshot of a consistent state of a set of design objects, after 1 or more semantically-meaningful changes.*

Figure 5.3 shows an example configuration, a password quality-checking program. It contains two programs, ck_pw and mk_db. As before, ck_pw has three components; mk_db has one. Both the ck_pw component ck_dict and mk_db use the component db, which is imported from another configuration, :dbm_lib:.

5.2.1 Consistency

A key characteristic of a configuration is that its objects are *sufficiently consistent* to serve the configuration's purpose; the states of the objects in it are internally consistent, and are consistent with each other.[2] "Consistent" must be defined in class- and application-specific terms.

[2] We are addressing consistency of the *product* — the objects developed and modified. We do not here address consistency of the *process* which yielded the product — *e.g.*, constraints specifying when a given transformation is legal, specifying the legal orders of transformations, specifying when a transformation should be applied automatically, *etc.* That important topic lies at the boundary between CCM and activities coordination / methodology support.

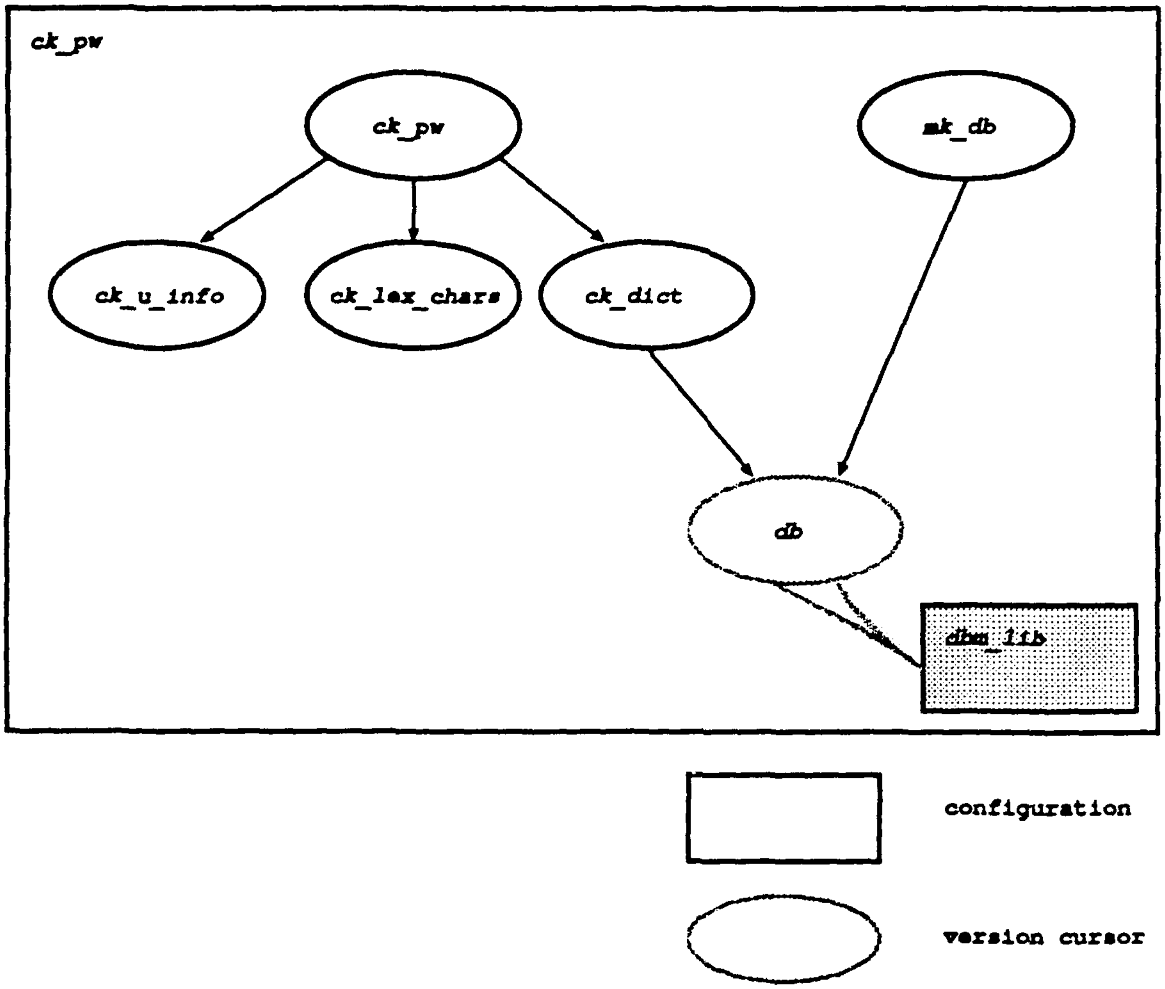

Figure 5.3: First version of Check-Password-Quality configuration.

5.2.1.1 Class-specific consistency

Most classes of design objects have their own unique definitions of consistency. For source-code objects, the programming language defines whether or not a given source-code object is a legal program. We may also choose to apply more stringent class-specific checks to a source-code object: a C program may be required to pass lint, an Ada program may be required to yield certain values when checked with metrics collectors, *etc.*

It is worth noting that a design object can be consistent according to its class-specific definition, but be erroneous. Various advances in the representation of design objects (*e.g.*, strong typing) have attempted to expand the degree to which consistency can be automatically checked; this additional formal structure, amenable to automatic checking, reduces the frequency of design objects which are formally consistent but erroneous in practice. In object-bases supporting user-defined constraints, constraints defined on classes are another form of class-specific consistency; careful definition of such class-specific constraints can be valuable in ensuring consistency.

An important type of class-specific consistency is *structural consistency.* Structural consistency considers the interaction of the interfaces of objects which are composed together. It is the consistency of the system's composition, and it is thus meaningful only in the context of a system of objects (ie, a configuration).

The degree of structural consistency-checking possible depends on how detailed we are able to specify the interfaces of objects. Consider relocatable-object files in a traditional operating system. The interfaces are "subprogram entry name" and "call name". It is possible to check for dangling calls, or uncalled entries, but it is impossible to check for type-mismatches in parameters or in return values, or for mismatches in the number of parameters. In a language like CLOS, VHDL, or Ada, the syntax of a module's interfaces — ie, the module's *protocol* — can be specified; a module's protocol typically includes the module's class, and the parameters, parameter types, and return value type of its operations. When such modules are composed, full *syntactic consistency* can be automatically verified. In experimental systems where the behavior of modules is formally specified, as well as their interfaces, then a degree of *semantic consistency* is subject to formal automatic checking. (See [11].)

Another important type of class-specific consistency, related to structural consistency, is *representation consistency.* For example, a derived representation (*e.g.*, relocatable-object file) should have a modification date greater than that of the primary representations from which it was derived (*e.g.*, the source file(s)). See Section 5.7.

5.2.1.2 Application-specific consistency

For any given configuration, the application domain determines additional consistency constraints which the configuration must satisfy in order to be sufficiently consistent to fulfill its purpose. Traditionally, these application-specific consistency requirements have been checked by the build-and-test cycle. For example, a password-changing program (Figure 6.8) should be tested to make sure that it rejects attempts to change a password without proper authorization, to make sure that it rejects attempts to mangle the password database via erroneous input, *etc.*

In systems supporting user-defined constraints on objects, such constraints can be very useful for automating part of the maintenance of application-specific consistency.

5.2.2 Composing Configurations

> Multiple donor configurations (*e.g.*, predecessor configurations, libraries of reusable modules) are typically composed in the process of creating or evolving a configuration. The entire donating configuration (all its objects) might be loaded into the recipient configuration; or, selected objects might be loaded into the recipient configuration; or, the *donating* configuration may be treated as a black box — certain of its external objects are referenced, but its other objects are hidden from the recipient configuration.

A configuration is typically built in part from other configurations; objects from other configurations are referenced, or are borrowed and modified. Typically there is one or more distinguished *predecessor configurations*, which are the immediate ancestors of this configuration. This configuration is a *successor configuration*, evolved from its predecessor(s); frequently it is considered to serve the same purpose(s) as its predecessor(s) — it is, in some sense, a "new version" of its predecessor(s). For an initial configuration, the predecessor configuration is the null (empty) configuration. This evolution of configurations creates version history DAGs (directed acyclic graphs) of configurations, related by ancestor/descendent relationships (see Figure 5.2).

Besides predecessor configurations, there will also typically be "donor configurations" from which objects will be referenced or borrowed — *e.g.*, libraries of reusable parts, pre-existing programs. In Figure 5.3, :dbm_lib: is a donor configuration.

Conceptually, there are several possible ways to access objects from a donor configuration:

direct reference The objects could be copied into the current configuration; in the current configuration, references to the imported objects are direct references to normal objects.

static cross-configuration reference In the current configuration, we could refer to the objects as they exist in the donor configuration, via static references. These static references might be direct references to normal objects, if the configurations share the same object space; more likely, they will be some sort of indirect reference ("object foo from configuration :C3.7:").

dynamic cross-configuration reference In the current configuration, we could refer to the objects via dynamic references which point to some version of the donor configuration, where the version is selected by some rule. Examples might include ("object foo from the most recent version of configuration :C: for Unix on Sun4"; "object foo from a version of :C: which is compatible with the version 9 IPC specification", "the highest-versioned object foo from my local workspace, if present there, else the highest version baselined").

Facilitating the composition of configurations and the reuse of pre-existing objects is important, but it leads to several complications:

Modularization and Interfaces Questions of modularization and interfaces — which are more typical in the programming language and software design contexts — must be considered in the design of a CCM model and system. See Section 5.2.3.

State clashes State clashes occur when a new configuration to be folded in contains objects which already exist in the current context, but the versions of the objects in the configuration to be folded in differ from the versions in the current context. For instance, a given application may depend on a particular version of an interprocess communication library; if it is advantageous to include a new module in the application, which depends on a different version of that library, the application and the new module may not be able to use the same library. See Section 6.2.2.

Dynamic references. A configuration in which all references to objects from other configurations must be static references is fairly inflexible; it has difficulty benefitting from improvements made to the configurations it references. Dynamic references allow the version to be selected according to some rule. See Section 6.4.

5.2.3 Modularization and Interfaces

> Questions of modularization and interfaces — which are more typical in the programming language and software — must be considered in the design of a CCM model and system. Reusing configurations is greatly simplified if configurations have clearly-defined interfaces, and support a distinction between external and internal objects, so that configurations can be used as black boxes. When abstractly specifying a configuration, it will be important to specify what objects the configuration *needs*, and what objects it *provides*.

The ability to define modules and define interfaces for those modules is traditionally regarded as the province of programming language design, but it strongly interacts with CCM. Configurations are composed in terms of how their interfaces interact. To adequately compose configurations, we must be able to represent, construct, and manipulate configurations in terms of the interactions of the interfaces of the components and utilities referenced.

CCM is thus helped greatly if the design objects' interfaces can be specified with considerable sophistication, so that objects can be referenced not just by name, but by class and/or protocol.

It is also useful to be able to specify the interface of a configuration. The specification of a configuration (see Section 6.3) should include the specification of:

- The objects which the configuration provides (exports) — its external objects.
- The objects which the configuration needs (imports) — its cross-configuration references (version cursors).

5.3 Design Transactions

> The specification and construction of configurations is a complex and difficult task which requires the effective cooperation of multiple agents. This task can be modeled by the notion of a ***design transaction*** — a long-duration sequence of operations, performed by multiple agents on a shared system of objects, which starts with one configuration ("check-out") and yields another consistent configuration ("check-in", or commit). Committing the transaction involves verifying the consistency of the resulting configuration; it adds a new node — a new configuration — to the history DAG.

A design transaction is a "semantically meaningful" change, which results in a state worth saving. It takes the set of design objects from one consistent state (ie, configuration) to another. (See [12].)

A transaction is a sequence of operations that appears to be one atomic operation, and which leaves the data it operates on in a consistent state. It is a sequence of changes which is one logical change. During the transaction, inconsistent states are allowed; a set of constraints are enforced at the beginning of the transaction, and at its end, but not during its lifetime. The transaction's internal inconsistent states must not be revealed — it must be a black box to agents not participating in the transaction. A transaction must be atomic in the face of concurrency and of failure. If two concurrent processes operate on shared data without protection, then the intermediate states of the data in one process may be observed by the other; transactions must prevent inadvertent concurrent access. Likewise, if a process halts abnormally because of a failure, the data on which it was operating may be left in an inconsistent state; committing a transaction must be all-or-nothing.

A design transaction frequently has an associated requirement, change request, or change notification (see Section 5.6) which it is meant to satisfy. The transaction functions as one logical change ("delta set"), encapsulating a complex series of smaller changes necessary to implement the logical change.

Multiple agents can participate in a design transaction. Because it is protected from outside access yet shared by a team of designers, a design transaction provides a sandbox (actually, a hierarchy of sandboxes) for cooperative work. Typically a large change to be performed to a system of objects is decomposed into smaller changes, and parceled out to multiple agents; nested transactions model such task decomposition. Nested transactions also support multiple levels of versioning and releases at various levels — to self, to group, to project, for unit testing, for integration testing, for distribution.

A design transaction goes through a cycle of initiation (check-out), work, validate, and publicize (commit, check-in). In a typical design transaction, a designer (or design team) initiates the transaction by checking-out a system of design objects from a central repository; typically a subset of some larger system of objects is checked out. Over a period of minutes, hours, or days, the designer(s) interact with the design objects via editors, transformers, analyzers, and validifiers. When the designer (or team) is satisfied with a system of design objects, the system of objects is checked-in again, for use by other designers.

But before any check-ins occur, the system of objects must pass a set of tests and checks for self-consistency. This validation process is typically complex and time-consuming, and is specific to the objects' classes and to the application. Both primary and secondary representations (see Section 5.7) are checked to the extent possible; the committing designer(s) are vouching that the primary representations are consistent with each other. The design assistance system should verify that the system of objects has passed the proper tests and consistency checks, before permitting the commit to succeed.

This process frequently occurs in a hierarchical fashion; a team checks-out a system of design objects, and then subsystems are checked-out of it by subteams. Normally subsystems are checked-out and checked-in (committed into the next higher level repository) repeatedly, as unit testing, subsystem integration, and subsystem testing occur; eventually the entire system will be checked-in (committed).

For example, consider the Check-Password-Quality configuration in Figure 5.3. Suppose that some improvements are required. Team A checks-out ck_pw and its direct components from a system repository, and places them in a team repository. After design decisions are made by the team, team member A1 takes responsibility for ck_u_info and ck_dict, and team member A2 takes responsibility for ck_lex_chars. They check-out those respective components from the team repository and into their personal repositories. Over a period of days they create new versions of these components with editors and compilers, checkpointing the states worth saving (via subtransactions) in their personal repositories. Various class-specific (*e.g.*, `lint` for C objects) and application-specific (*e.g.*, test suites) methods are used to verify the sufficient consistency of these versions. At some point it must be verified that ck_u_info and ck_lex_chars will cooperate, so A1 and A2 check them into the team repository, and perform consistency checks on the whole configuration. Eventually, all the objects in the team members' repositories have been checked into the team repository, and the team's configuration has been determined to be consistent; at that point, it can be checked into the system repository.

5.3.1 Design Transactions vs Conventional Transactions

Design transactions therefore differ in several ways from traditional database-oriented transactions.

volume, duration, complexity, scope Traditional transaction processing is characterized by high-volume, short-duration, and simple transactions which access only a few records. Design transactions are relatively low volume, long-duration, highly complex, accessing large data structures, and are typically distributed over a network.

consistency Traditional transactions define correctness in terms of serial consistency. But for design transactions, the data itself, rather than the order in which it is accessed, actually determines consistency. Design data must be self-consistent, and sufficient consistency is defined by the objects' classes, and by the particular application.

permanence The completed results of traditional transactions must survive a system crash. The individual operations within a design transactions should do the same — designers require that as much of their work as possible be saved if a crash occurs. It is therefore desirable to bring the object-base back to it latest possible state, which need not be configuration-consistent (in the middle of a transaction), but which should be filesystem-consistent. But where long-running conventional transactions are usually aborted on system restart, long-duration design transactions are not; only the lower-level operations within the design transaction are subject to abort on system crash. Further, an additional permanence requirement is placed on the results of the design transaction itself: the results of a design transaction should by default persist, and not be overwritten by newly-committed consistent data. The data operated on by a design transaction must be history-bearing.

atomicity Design transactions should be concurrency-atomic (providing isolation) and failure-atomic (providing recovery). However, unlike traditional transactions, in design transactions there are often several agents participating in a transaction; the transaction's internal states must be visible to these agents, but not to any others. (The agents typically further partition the transaction via subtransactions, but they must have access to the super-transaction to commit their subtransactions into it, at integration time.) Also, as noted above, long-running conventional transactions are typically aborted on system failure, but long-running design transactions must survive a system failure (though individual operations within them may be aborted, their effects erased).

5.3.2 Validation at Commit Time

When the attempt to commit a design transaction is made, the design system should verify that the configuration is consistent, according to class- and application-specific definitions of consistency (see Section 5.2.1). The painful traditional way to do this is "build and regression-test". In systems where the interfaces of modules can be specified with some sophistication (*e.g.*, CLOS, VHDL, Ada), syntactic consistency — the consistency of how the interfaces of the objects are composed with each other — can be automatically checked. Similarly, if behavioral specifications of the modules are available, some formal semantic consistency-checking can be performed. Where V&V checking tools are available — *e.g.*, in many CAD environments — those tools should be integrated into the validation-at-commit process. Similarly, user-defined constraints on objects and classes should be re-checked. Representation consistency should also be checked.

5.4 Equivalence and Search

> A recurring theme in the CCM problem is the issue of equivalence and compatibility. Certain objects are equivalent to one another with respect to particular operations in particular circumstances; a central task of CCM is to know — to track or to determine — which objects are equivalent, with respect to particular operations (*e.g.*, link, compile) in particular environments (*e.g.*, "SunOS 4.0.3c using the DNS-based C runtime library", "Allegro LISP with PCL 12/88 loaded"). CCM can be thus cast as a search problem: locate an appropriate object (or system of objects) to satisfy the current need in the current context.

CCM must deal with multiple states for the same object (X:1.1, X:1.2:, X:2.0), and multiple representations for the same conceptual object (uiv.spec, uiv.lisp, uiv.exe). Thus, CCM rapidly leads to issues of object identity.

Versions, equivalence, and compatibility Consider X:1.1:, X:1.2:, and X:2.0:. Stating that they are all versions of X is to state a historical fact (they all derive from each other, or from a common ancestor). It also implies that they are useful for the same purpose, to some unspecified extent. We want to distinguish among these three states of X — we must be able to speak of them individually, and other objects must be able to refer to them individually — to particular versions of X. We also want to consider them to be, in some sense, "the same object". They are different versions of "the same thing", and we sometimes must be able to refer to X, independent of version, in our speech and in our object-base. (X is in a sense a handle for the entire *version set* of X:1.1:, X:1.2:, and X:2.0:; X may be considered a *version-generic* reference.)

Representations, equivalence, and consistency When we have multiple representations for the same conceptual object, problems of identifying and maintaining equivalence between representations arise: we need to identify which objects are actually representations for "the same thing", and we need to maintain this relationship in the face of change. If the specification for uiv, uiv.spec, changes, then uiv.lisp and uiv.exe are probably no longer consistent with it — they no longer represent the same conceptual object. If someone modifies uiv.lisp, making uiv.exe consistent with it is a fairly straightforward matter of compilation. But making uiv.spec consistent with it is more difficult to do and to verify.

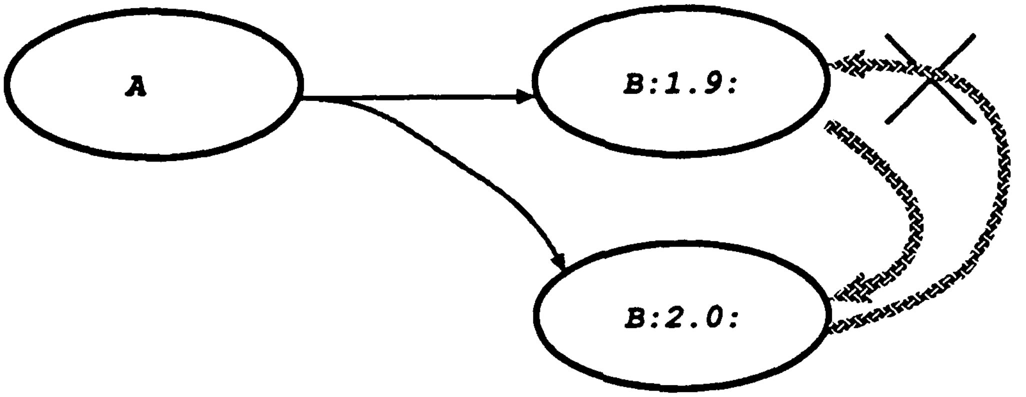

Figure 5.4: A expects B:1.9:, but can accept B:2.0:.

5.4.1 Versions, Equivalence, and Compatibility

Objects X:1.1:, X:1.2:, and X:2.0: share a common identity, X. We may call this the ***signature*** of X. The signature is minimally a name; more usefully, the signature includes a protocol (ie, the object's class, specifying a set of applicable operation interfaces, including typed parameters and return types). Some experimental systems (*e.g.*, [11]) include a behavior specification of the operations; we can thus formally determine if the objects behave the same in a given context. But properly producing such specifications so that they are complete and consistent is difficult; differences in the behavior of two versions are frequently discovered by testing and experience, and can be modeled as compatibility differences.

Two states of an object (or system of objects) are ***version equivalent*** if they are interchangeable without surprises. ***Version compatibility*** is a weaker form of equivalence; we speak of upward compatibility, strict compatibility, *etc*.

Version equivalence and compatibility are meaningful only the the context of a particular desired operation in a particular environment. For example, we may state that "X:1.1:, X:1.2:, and X:2.0: are binary-compatible in the SunOS 4.0 environment" — that is, they are compatible with respect to the linking operation in that environment. They may not be compatible with respect to the compilation operation (ie, source-compatible).

Compatibility is frequently directional. For instance — assume that the context is some operation op in some environment env. A common situation is that if A expects B:1.9:, then it can accept B:2.0: (Figure 5.4); but if it expected B:2.0:, it cannot accept B:1.9:. The ***satisfaction set*** of B:2.0: with respect to [op, env] is a superset of the satisfaction set of B:1.9:; B:2.0: satisfies the same requests (under [op, env]) as

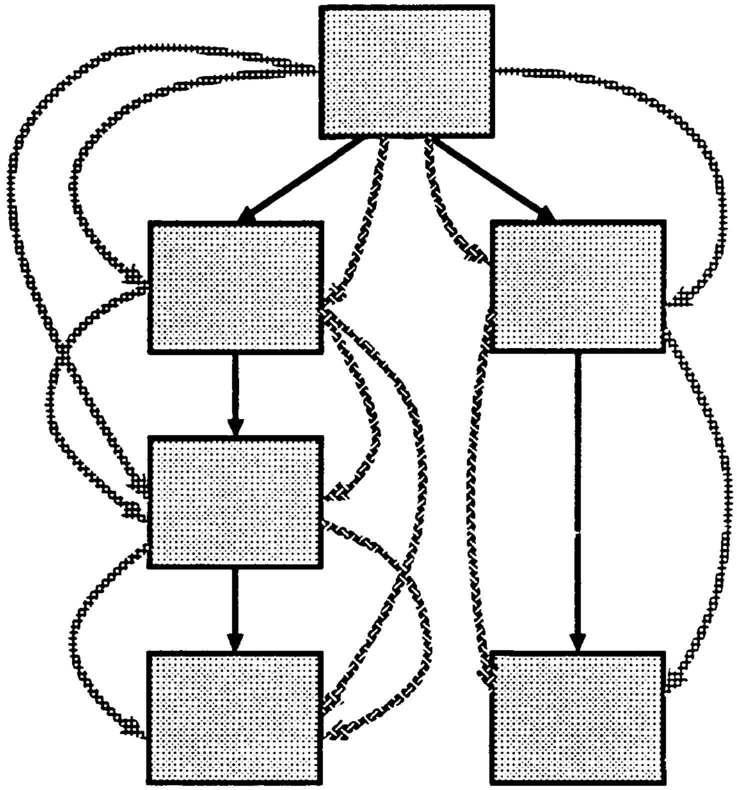

Figure 5.5: Satisfaction DAGs indicate compatibility.

B:1.9:, and optionally more requests. If the two satisfaction sets are identical, then the two objects are version-equivalent (with respect to [op, env]).

Version equivalence and compatibility can be seen as defining a set of overlapping *satisfaction DAGs* (Figure 5.5). Each satisfaction DAG identifies versions which are compatible with respect to a particular operation in a particular environment. The nodes of a satisfaction DAG are nodes from a version history DAG; a satisfaction-DAG arc from node B:1.9: to node B:2.0: indicates that if B:1.9: is desired, B:2.0: can be substituted for it. That is, an arc from B:1.9: to B:2.0: means that B:1.9:'s satisfaction set is a subset of the satisfaction set of B:2.0:. A bidirectional arc thus implies equivalence.

One peculiar operation under which versions can be compatible is "cohabitation"; this is really an indication of the degree to which the two objects are *in*compatible. Incompatible versions are sometimes cohabitative — that is, they can be referenced in the same configuration; but sometimes incompatible versions are not cohabitative — they are antagonistic. Consider an application which depends on a particular version of an interprocess communication library; if it is advantageous to include a new module in the application, which depends on a different version of that library, it may be that neither version of the library will satisfy both the existing application and the module to be reused (an unresolvable state clash). If the two versions of the library are cohabitative in that environment, then the new module can continue to use its own version of the library, referencing it indirectly via a version cursor. But if the two libraries contend for the same device antagonistically, then the same configuration cannot make use of both.

5.4.2 Representations, Equivalence, and Consistency

For purely mechanical transformations (Section 5.7), *representation equivalence* can be deduced from the transformation history, since we assume that purely mechanical transformations yield equivalent representations. But if we have two primary representations, or two representations which are related by a transformation which requires human input, then consistency must be checked by some verification procedure.

5.4.3 Search

The use of dynamic references leads to issues of search — we must locate an object definition which satisfies the reference. The search may be trivial or significant, depending on the generality of the version-selection rules. When the rules are sufficiently general, backtracking becomes a necessary capability, in order to find a consistent set of versions. For example, assume object A references pcl with a version-selection rule which would be satisfied by pcl:7: or pcl:8:; object B does likewise, but can only accept pcl:8:; and the two versions of pcl cannot coexist. If we dereference A's pointer to pcl with pcl:7:, we'll need to back that dereference out when we attempt to dereference B's pointer to pcl.

5.5 Dynamic Version Binding

> When specifying a configuration, we typically want to refer to versioned objects. We may refer to a versioned object by a static reference which denotes a single definite version, fixed at specification time; or we may refer to a versioned object with a *dynamic reference* — a reference which will be resolved to a particular concrete version later, typically at configuration-construction time. Underpowered dynamic references point to a "default" version of an object, where the default changes as the object evolves. Fully general dynamic references function as search rules for locating an appropriate object, based on the current operation and environment.

When we refer to a versioned object (an object which may have multiple versions), we may refer to it statically or dynamically ([13]).

A configuration in which all references to versioned objects must be static references is fairly inflexible; it has difficulty benefitting from improvements made to the objects it references. For instance, assume configuration :Tcheck: uses version 10.1 of X-windows; suppose all references to objects from X-windows are static references to objects from 10.1. If a new version of X-windows — say, X-windows:11.4: — becomes available, then Tcheck, because of its static references, still uses the objects from X-windows:10.1:. Upgrading Tcheck to use X-windows:11.4: may be tedious and time-consuming. But if Tcheck had referenced objects from X-windows:default-configuration: or X-windows:"binary compatible with Tcheck's graphics calls":, then the upgrade might occur semi-automatically.

Underpowered dynamic references point to a "default" version of an object, where the default changes as the object evolves. For instance, on VMS, aardvark.exe;5 refers to a specific concrete version; aardvark.exe without a version number is a dynamic reference, denoting the highest numbered version; aardvark.exe;-1 is another dynamic reference, denoting the version before the highest-numbered one; *etc.* This notion of a "default version" or "current version" associated with a version set is a weak idea — the presence of multiple alternates means that multiple "default versions" are current; which of the alternates is actually "default" depends on the current environment and the operation desired. Fully general dynamic references function as search rules for locating "the right" object, based on the current operation and environment. Examples might include X-windows:"binary compatible with Tcheck's graphics calls":; foo:"the highest-versioned from my local workspace, if present there, else the highest-versioned baseline":.

Dynamic references serve several purposes:

- They facilitate object reuse.
- As mentioned above, they facilitate automatic upgrades — when a newly-created version will satisfy a dynamic version cursor previously satisfied by a different version, the user of that cursor can be automatically notified (see Section 6.3.1).
- They make it possible to conditionalize construction and consistency-establishing operations on the environment and on the desired operation. For example, in Figure 6.8, npasswd references one of two possible be's; be_{fs} (filesystem-based back end) or be_{yp} (daemon-based back end) . Selection of the appropriate be can depend on the environment for which npasswd is being built.
- They allow references to be made to objects which are not currently available — "hypothetical objects" which are being created or will be created in a foreign configuration. In such a circumstance a version cursor is a specification for an object whose existence is requested — "put here some A which meets these constraints". Version cursors thus enable a sort of "lazy integration", where components and utilities are found and chosen as they are needed.

A version cursor is a request for an object. Unresolved version cursors in a configuration are analogous to unresolved external references in a relocatable-object file. The version cursors are references; the definitions must be found elsewhere, in other configurations. Resolution must occur when the configuration is constructed (made consistent), analogous to linking the relocatable-object files.

Version cursors are typically resolved when configuration construction occurs (*e.g.*, when derived representations are generated, when a request is made to produce a concrete configuration).

Version cursors must specify the object in such a fashion that it can be located. Since a version cursor must resolve to a particular version of an object, it must specify:

The object's signature. This may be a name, an interface or protocol specification, or something more complex.

A version-selection rule. This may be "version :1.5:" (for a static cross-configuration reference), or a more-or-less complex dynamic reference. The form of version-selection rules depend heavily on the form of compatibility attributes (see Section 6.1.2).

Note that with a dynamic version cursor, the version which is referenced at can change in two ways:

1. The version cursor's selection rule may be manually changed — *e.g.*, to capture an improved understanding of the requirements the object must satisfy ("A must satisfy the version 9 IPC spec, *and* it should have responded to change request 1197").

2. The version referenced may change as a side-effect of changes in the object base — *e.g.*, a new version has been checked in which satisfies the selection rule.

5.6 Change Propagation and Change Notification

> When a design object changes, objects which depend on that object may need to react to that change, to re-establish consistency; dependent objects might update themselves, update other objects, or notify a human. When an object changes, dependent objects must be notified, and consistency re-established, but the percolation of change must be controlled — every object which needs notification should be notified, but we must avoid notifying every object in the object-base at every change.

When X changes, objects which depend on X — *e.g.*, Y — may need to react, to re-establish consistency. It may be necessary to propagate change along many types of relationships. When a component changes, it may be advantageous to generate a new version of the assembly; when a utility changes, its users may need to change; when one representation changes, it may be advantageous to generate other representations from it; when a predecessor version changes, it may even be advantageous to fold the change into a descendent version.

A CCM system should provide some automated means for detecting that the dependency between X and Y has an change which has not been responded to. We may choose to use a passive, or flag-based, strategy for handling this: the problem is noticed when Y is next accessed. This is the strategy used by Make — if the source-file object has changed, there is no automatic action regarding the executable object; but the situation is detected at the next build, and the executable is regenerated. Alternately, we may choose an active, or message-based, strategy — messages ("change notifications") are sent to the dependent objects, which choose how to react to the message.

Change propagation should be integrated with the standard mechanisms by which change — both human-instigated and automatic — is managed by the object-base. For instance, change notifications can be considered a subclass of change requests — they are change requests produced by other objects rather than by humans. It is worth noting that, while subtransactions are useful to model top-down change, change notifications often serve the purpose of modeling bottom-up change.

A key problem is limiting the propagation of change while still notifying every object which requires notification. It is sometimes reasonable to prune the propagation manually — ie, by requesting that the user interactively limit the area affected. Alternately, it has been proposed in [14] that by placing attributes on relationships (ie, slots), we can correctly restrict the propagation of operations, or of change ([15], [16]).

But marking a relationship as being always insensitive to change is a major assertion. Determining whether to propagate a change depends not just on the characteristics of the slot, but also on the purpose of the change. The change-sensitivity markers would need to be fairly sophisticated expressions, conditionalized on several factors, including the operation and the environment. The change-sensitivity markers would often need to be specialized on an object-by-object, rather than just a class-by-class, basis.

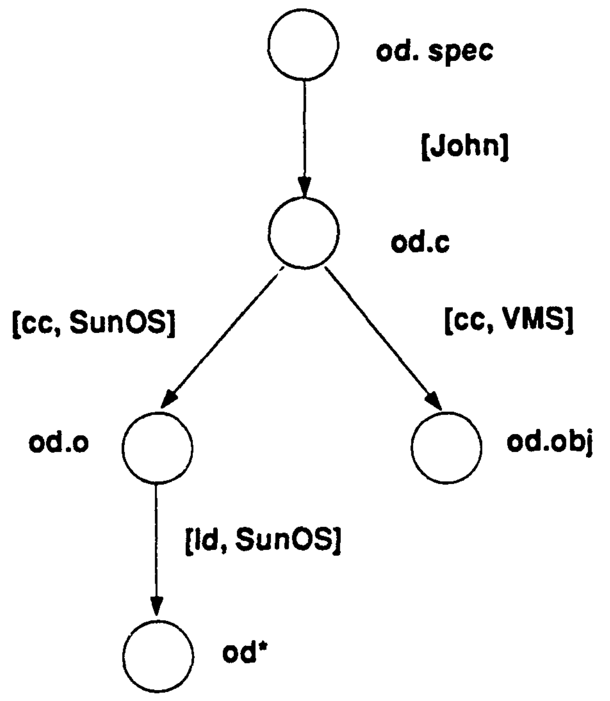

Figure 5.6: Multiple representations for the same design object.

5.7 Objects with Multiple Representations

> Design objects are frequently multi-representational. The same conceptual object is depicted by several different representations, often at different levels of abstraction. For example, an ALU hardware design may be represented by a layout object and a netlist object; a program may be represented by a spec object, a source code object, and an object-file object. Representations are related to each other by transformations. A *primary representation* is produced with human input. A ***derived representation*** is generated from another representation by application of a transformation (*e.g.*, the compile transformation, which derives relocatable-object from source-code); if the transformation is purely mechanical, the derived representation is a ***secondary representation***. Multiple representations and the transformations between them must be managed in order to track and maintain consistency.

Design objects are frequently multi-representational. The representations are typically rich highly-structured objects, which may include text and graphics. Consider a specification for a program, od.spec (Figure 5.6). From the spec, a programmer might derive C source od.c (or Lisp source, od.lisp). From either of these, a compiler might derive an object file — od.o from a Unix compiler (for a Unix environment), or od.obj from a VMC compiler (for a VMS environment). We'd like to consider all of these od's as representations of "the same conceptual object". We may therefore talk about a ***representation set***: a set of objects which we know are multiple representations for the same conceptual entity.

When we have multiple representations for the same conceptual object, problems of identifying and maintaining equivalence between representations arise: we need to identify which objects are actually representations for "the same thing", and we need to maintain this relationship in the face of change.

Representations are related to each other by transformations. The transformation may be versioned (*e.g.*, ada.exe:6.0:), and its actions may be dependent on the environment. The relationship between two representations is thus defined by a [operator.version, environment] pair. The representations in the representation set form a representation DAG, where the arcs are these transformation relationships.

A transformation may involve human input; in the [operator.version, environment] pair, the operator may be mechanical, human, or both. For example, traditional compilation is mechanical; the derivation of source-code from program-specification is typically human; and a human-assisted optimizer would be a human/mechanical mix. Purely mechanical transformations yield secondary representations; the others yield primary derived representations.

A representation whose creation or modification involves human input is a *primary representation*. Primary representations must be checked for correctness, and for consistency with other primary representations. *Secondary representations* are generated from other representations (primary or secondary) by the application of a purely mechanical transformation.

A ***derived representation*** is any .epresentation which is derived from another representation by the application of a transformation. A derived representation may be a secondary representation (if the transformation is purely mechanical), or it may be a primary representation (if the transformation involves human input).

- An underived primary representation is created purely by humans, not derived from other existing representations.
- A derived primary representation is a representation which is derived from some other representation by a human or human/mechanical transformation.
- A derived secondary representation is one which was derived via a purely mechanical transformation; underived secondary representations cannot exist.

Frequently there is no clear answer as to whether a particular representation must be treated as a primary representation or as a secondary representation. For instance, suppose we decide to allow patching of relocatable-object objects. We may choose to make relocatable-object a primary representation; modifying a relocatable-object would cause a change notification (active or passive) to be posted on the corresponding source-code object. Or, we may define a new transformation, patch-object, which takes as input a relocatable-object and a patch script (which is a new primary representation).

In all purely mechanical cases, we assume that the operator guarantees correctness of the secondary representation which the operator derives. We also assume that a purely mechanical operator guarantees consistency between source and derived representations — if the same [operator.version, environment] is applied to a particular representation, an identical result representation will be produced. Transformations which are not purely mechanical yield derived primary representations which must be checked for correctness, and for consistency with the source representation; this is particularly hard when the target representation is at a higher level of abstraction than the source.

Primary representations should be history-bearing objects; a change to a primary representation may cause change propagation effects (*e.g.*, the notification of objects which have version cursors referencing that primary representation.) Secondary representations, since they can always be re-generated from their corresponding primary representations, need not be history-bearing; the creation of a new secondary representation need not cause change propagation effects (though it is often useful for it to do so).

The traditional approach for maintaining consistency among multiple representations is to assume that change only occurs in one particular primary representation; all other representations are (transitively) derived from that one in a purely mechanical fashion — ie, they are all secondary representations. Consistency can then be established by simply re-deriving all the secondary representations. For instance, we may allow changes to source-code objects, but editing the relocatable-object or executable is considered bad practice; they must be re-generated from the source-code. This becomes complicated when there are multiple primary representations (*e.g.*, program spec and program source-code), when there are significant derived primary representations, and when a primary representation is at a lower level of abstraction than a derived representation (since going in that direction — *e.g.*, generating source-code from relocatable-object — is difficult).

Automated change notification (see Section 5.6) can help. In general, we may choose to track changes, automatically regenerating secondary representations when appropriate, and providing interactive tools to inform the designer of changes which must be made — *e.g.*, to regenerate derived primary representations.

Chapter 6

KBSA Framework CCM Model

This chapter summaries the CCM model for the KBSA framework. It also describes the experiences of a project which is using the model in an avionics CAD domain.

6.1 Configurations as Contexts for State

> CCM is a global issue; it cannot be dealt with locally, on an object-by-o^bject basis. In particular, it does not make sense to create a new version of an object by copying the object, because it is frequently impossible to halt copy-propagation before a large fraction of the object-base is duplicated. If objects may have multiple states, data access by (object, slot) no longer makes sense; all references to the state of an object must be made *in the context of some configuration.* Every data access must specify not just the object and slot, but also the configuration in which the access is to be performed: (object, slot, configuration). Thus, in our model, a configuration is a repository of state — it holds the state of the objects occurring in it, mapping object × slot → value.

For CCM, we must determine how to represent change, given that

- The objects which are changing are interdependent.
- We must support both forward and backward changes (recoverability).

Many models and systems represent change by copying. When a version of an object is checked-in, that copy of the object becomes read-only. When there is a need to

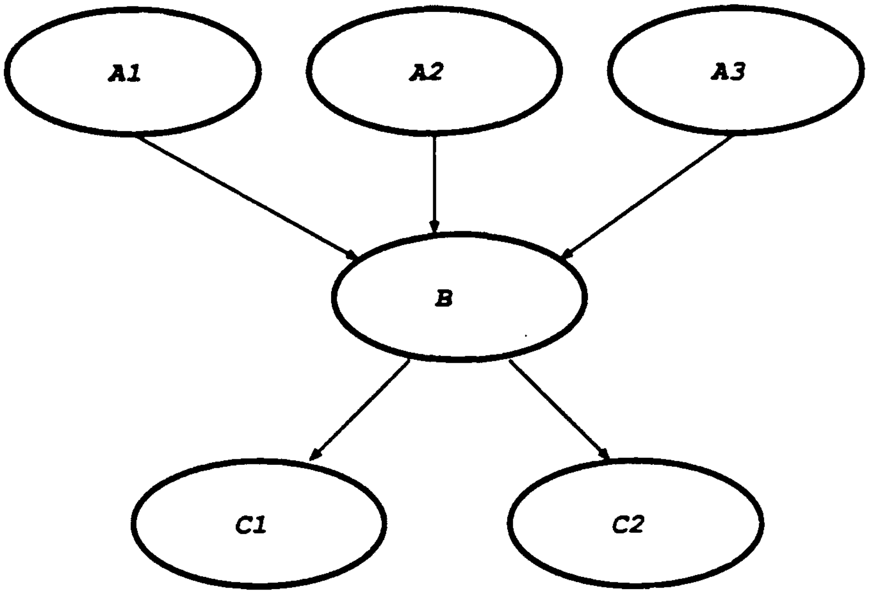

Figure 6.1: Changing a dependent object.

modify the object (culminating in a new version), a writable copy "of the same name" is created (check-out). This st:ategy works reasonably well in the domain of files and filesystems; but in highly interconnected and interdependent object-bases it quickly becomes unmanageable, due to issues of object identity.

1. Consider the object being copied. When we copy an object, we do not necessarily want to simply duplicate the object references within that object (*shallow copy*); we may need to duplicate the objects referred to by that object, and have our new object refer to those duplicates instead (*deep copy*).

   Suppose cm.tex (Figure 4.1) was previously checked-in, and we now want to modify it — ie, we need a new version. If we represent change by copying, then we do not want just a copy of cm.tex — we want its components, too. For instance, if we should decide that we must modify cm.tex's component exper.tex, we don't want to modify the same exper.tex referenced by the checked-in version — we would be modifying the checked-in version simultaneously, as a side effect. Thus, when we request a new version of a hierarchical assembly of parts, we want not only a copy of the root, but of all its *components*, recursively; alternately, we wish those component references to be marked copy-on-write.

   Because the *.ps objects are transitive components of cm.tex, we will want copies of them, too. But suppose that the *.ps objects included a slot which identified the drawing program which produced them. We do not (normally!)

want to cause new copies of the drawing program to come into existence when the *.ps objects are copied; we want the new *.ps objects to refer to the same drawing program as the old copies. Thus, we want some slots to obey deep-copy semantics, but some to obey shallow-copy semantics.

2. More problematically, consider not the object being copied, but the objects which refer to it. The slots which previously pointed to the older copy *may* need to be readjusted to point to the writable copy, and the objects where those references occurred *may* themselves need to be copied (since they are being implicitly changed via changing objects which are in their slots). This is the ***change propagation*** problem (see also Section 5.6).

   In Figure 6.1, A1, A2, and A3 reference B. B references C1, C2, and C3. Assume that B has been checked-in. We decide to change B — for instance, by changing its reference to C1 to be a reference to C4. We must make a new copy of B in which to make the change, so that we maintain history (the state of B had been checked-in). But B is a feature of A1, A2, and A3, and we've changed that feature; they have therefore changed, so perhaps we should make a new copy (a new version) of them, too, pointing at the new B. Likewise, we may need to copy the parents of A1, A2, and A3, and their parents, transitively.

It has been proposed that by placing attributes on relationships (ie, slots), we can correctly restrict the propagation of operations such as copy (see Section 5.6). But determining whether we need to copy the objects which refer to an object which has changed cannot be determined solely from slot attributes; it depends not only on the characteristics of the slot, but on the purpose of the operation. Frequently, criteria which can be used to limit the recursive copy propagation without sacrificing correctness and consistency are not readily apparent.

Consider figure 6.2. In configuration :C1:, we have (SlotA A) and (SlotB B) pointing to X, and (SlotX X) pointing to Y. Configuration :C2: has elected to reuse X (and thus Y). Suppose that, in :C1:, (SlotX (SlotA A)) is changed, to point at Y'. The X referred to by A has changed. Should B refer to the new X or the old X — what, if anything, should happen to (SlotB B)? Further, to which X should D refer — what should happen to (SlotD D)? The right choice depends on what the owners of B and D intended when they referenced X.

Because of these problems, our model does not represent change by copying. We contend that if an object is to exist in more than one configuration — ie, have more than one version — then all references to the state of an object must be made in the context of some configuration. If objects may have multiple states, data access by (object, slot) no longer makes sense. Data access must also *in all cases* specify

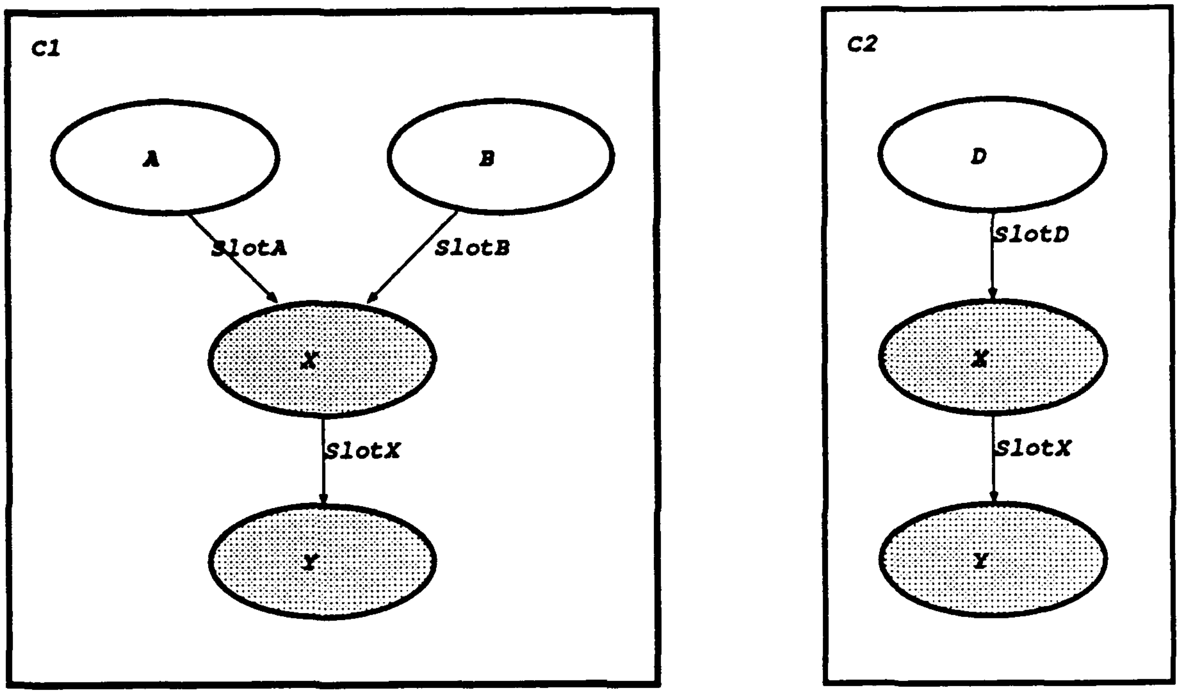

Figure 6.2: Reusing a dependent object.

the configuration in which the access is to be performed: (object, slot, configuration). This specification of a configuration can be implicit, but it must occur.

Therefore, in our model, we consider a configuration to be primarily a context for determining the state of objects; it is a repository of state. A configuration records the state of the objects which are mentioned in it, not the objects themselves; it maps maps object × slot → value. It can be considered to be a table of (object_id, slot_id, *value*) tuples.

We are thus versioning *sets* of objects — configurations — rather than individual objects. If object A exists (is mentioned) in configuration :C:, then the state of A in :C: is called a ***version*** of A; object A in a different configuration is a different version of A. Versions of objects exist only in configurations; objects under CCM have no state outside of a configuration. This gives us the same semantics as deep copy, but with a space-efficient representation, and a conceptual simplicity.

In some ways, a configuration is thus analogous to a scope in a block-structured language or a package in an information-hiding language. A successor configuration imports state from its predecessor configuration; the effect is similar to importing objects from a package, except that the imported objects obey copy-on-write semantics. A subtransaction (see Section 6.2.1) is similar to a nested scope — objects in the

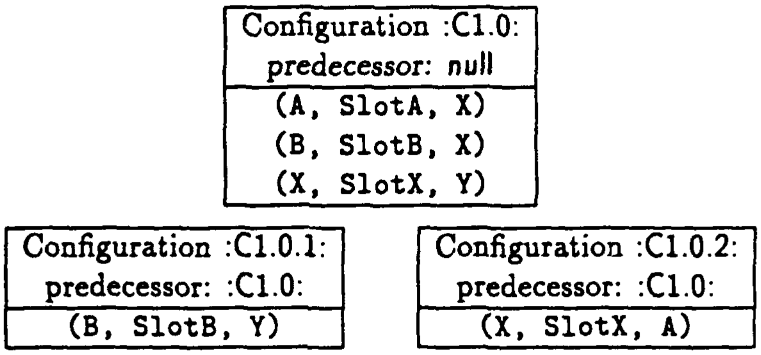

Figure 6.3: Configurations :C1.0.1: and :C1.0.2: as deltas from :C1.0.1:.

super-transaction are visible, but may be hidden by local redefinitions (changes).

6.1.1 Configurations as Deltas

> A configuration need only record the *changes* in state which were made during one design transaction — a *successor configuration* is a delta from its predecessor. This leads to a space-efficient representation of a tree of configurations; with appropriate design, the representation can also be time-efficient (*a la* RCS [10]).

A history DAG is a directed acyclic graph of configurations which share a derivation history; the configurations are related by ancestor/descendent links. In our model, we restrict the history DAG to be a tree — ie, we disallow multiple predecessors (parents); we model multiple predecessors as one predecessor plus other donor configurations "loaded in". Because of this, a configuration need only record *changes* in state — a successor configuration is a delta from its (single) predecessor.

In Figure 6.3, we see configuration :C1.0: (which is :C1: from Figure 6.2), and its child configurations, :C1.0.1: and C:1.0.2:. We assume that :C1.0.1: was produced by

1. Checking-out :C1.0:, and
2. Making one change: (setf (slotB B) Y).

and that :C1.0.2: was produced by

1. Checking-out :C1.0:, and

2. Making one change: (setf (slotX X) A).

Only the *changes* need to be recorded in the child configuration. When it is necessary to read the state of unchanged objects, we traverse the predecessor link (recursively, if necessary). So, for instance, in :C1.0.1:, the value of (SlotB B) is Y, and the value of (SlotA A) is X (retrieved from :C1.0:); in :C1.0.2:, the value of (SlotX X) is A, and the value of (SlotB B) is X. This gives the same semantics as deep copy, but with a space-efficient representation.

This example uses "forward diffs" — the predecessor is treated as the original, and the successor as the revision. For a representation which is not only space-efficient but also time-efficient on average, "backward diffs" can be used as in RCS: the successor stores the complete state of the system of objects, and the predecessor's table indicates the changes which must be made to the successor to restore the predecessor.[1]

In version-control systems like RCS, the version history of a file is recorded in a single file; from that version-history file, any version of the file under revision control can be extracted. Versioning is thus done on a file-by-file basis, analogous to modeling change by copying. Our configurations would be similar to version history files maintained on *systems* of files, rather than on individual files; they would be, in effect, a combination of "patch" files — which describe deltas on an entire system of files — and RCS ",v" version-history files.

6.1.2 Compatibility Attributes

> Besides objects' state, configurations also include *compatibility attributes*, which annotate the history DAGs with *satisfaction DAGs*; this information is used when de-referencing version cursors.

Determining equivalence and compatibility of versioned objects is a central CCM task (Section 5.4). Version equivalence and compatibility must be considered in the context of a given operation in a given environment — pcl-v7 and pcl-v8 may be source-compatible (compatible with respect to compilation), but not binary-compatible (compatible with respect to linking). A degree of equivalence and compatibility information can be determined formally, from the structure of objects; for instance, two objects with radically different protocols are not likely to be equivalent. But in general, testing and experience must be used to make such determinations. The inability

[1] A production CCM system would also have to address compaction and garbage collection issues. It would be useful to have the capability of occasionally identifying configurations in the history DAG which are no longer worth saving; these could be compacted out of the DAG, by migrating their changes into preceding or succeeding configurations.

to retain and use such information (including automated use) is a major limitation of existing systems. *Compatibility attributes* can be set on configurations to record the knowledge gleaned from such testing and experience. In our model, compatibility attributes are part of the data making up a configuration.

Compatibility attributes define the set of satisfaction DAGs (see Section 5.4). A compatibility attribute can be considered to be at least a labeled directed arc between two configurations. The label specifies the operation and environment under which the two are compatible, and the direction specifies the subset relationship between the two configurations' satisfaction sets.

Dynamic version cursors make significant use of compatibility attributes in specifying and resolving version-selection rules; see Section 6.4.

Compatibility depends on the environment and the desired operation. Compatibility information must include history information about the state of the configuration when a (versioned) transformation was performed. We believe that the properties and transformations should be structured via inheritance (*e.g.,* gcc:1.35: is a kind of gnu C-compiler, which is a kind of C-compiler). The exact format of the compatibility information, the model of incompatibility, and the format of version-selection rules which use such information needs to be further determined.

6.1.3 Structure and Operations of Configurations

Figure 6.4 shows the basic structure of a configuration, and three of the basic operations, make-instance, read-slot and write-slot. We assume that classes are defined elsewhere. We also assume that objects are defined locally in a configuration, though they need not be.

```
Configuration            ::= < State-table, Compatibility-attributes, Symbol-table >
     --- A configuration is a state table, compatibility attributes, and
     --- a symbol table.

State-table              ::= { State-tuple* }
State-tuple              ::= < object-id, Slot-id, Value >
Value                    ::= atomic-value | object-id
     --- The state table contains state-tuples.

Compatibility-attribute  ::= <operation-id, environment-descriptor, configuration-id>
     --- A compatibility attributes indicate another configuration which
     --- is compatible with this one.

Symbol-table             ::= { Object-descriptor* }
Object-descriptor        ::= <object_id, class_id>
     --- The symbol table indicates the class of an object.

(make-instance Class Configuration)
   ;;; makes an object by entering it into the symbol table of
   ;;; Configuration as an object of a particular class.

(read-slot Object Slot Config)
    if there-exists state-tuple ST in Config such that
            (object-id ST) = Object and (slot-id ST) = Slot then
        (value ST)
    else if (predecessor Config) = NULL then
        <no such object>
    else
        (read-slot Object Slot (predecessor Config))

(write-slot Object Slot Config Value)
    if there-exists state-tuple ST in Config such that
            (object-id ST) = Object and (slot-id ST) = Slot then
        (setf (value ST) Value)
     else
        (insert-into (State-table Config)
                     (make-state-tuple Object Slot Value))
```

Figure 6.4: Primitive structure and operations for configurations

6.2 Transactions as Contexts for State

> The initiation of a design transaction creates a *transaction handle* or proto-configuration. The transaction handle is a work context; it is a writable configuration — a repository for the state of a set of objects under CCM. Transaction handles are long-duration, sharable, and atomic; they may be nested to provide a hierarchy of workspaces. When the transaction is successfully committed, the state of the transaction handle becomes read-only: the transaction handle becomes a configuration, guaranteed to be a consistent system of objects, based on class- and application-specific definitions of consistency

A design transaction is an atomic, long-duration sequence of operations, performed by multiple agents on a shared system of objects, which starts with one configuration ("check-out") and yields another consistent configuration ("check-in"). Basic design transactions are discussed in Section 5.3. In the KBS.\ framework CCM model, the operations making up a design transaction are performed on a transaction handle — a writable proto-configuration. The transaction handle is a first-class object, which can be passed around and shared. The handle is shared by the agents performing the transaction; users who have its handle, and write-permission to the handle, can perform operations within the transaction. When the transaction is committed, the transaction handle is checked for consistency; if it passes the consistency-checks, the transaction handle becomes a configuration — its state table becomes read-only.

Committing a transaction causes the proto-configuration's dynamic references to become fixed; all dynamic references within the configuration's state table become static references. All components and subcomponents become fixed, so that the configuration could be described by a linear list of particular versions of objects (by specifying subcomponents, we have constrained the components not to change). However, the configuration schema contains the unfixed version cursors, allowing the abstract specification of the configuration to be reused in a descendent configuration; see 6.3.

Though the state-tuples in its state table are frozen, certain modifications can be made to a configuration after commit.

- Compatibility attributes can be added, modified, and deleted, to record the results of testing and experience. Similarly, change-request annotations can be added, describing bugs, deficiencies, and wish-lists.

- It may be possible to add new secondary representations, depending on the purpose of the configuration. If the configuration is only for change management, then new secondary representations can be allowed. If the configuration is intended to be a release for a particular platform, the presence or absence of secondary representations is significant, and the addition of new secondary representations will probably be disallowed.

6.2.1 Successor Transactions and Subtransactions

The DAG of configurations grows as design transactions are applied to existing configurations (Figure 5.2). A design transactions which starts with an existing configuration and yields a successor configuration (see Section 5.2.2) can be called a successor transaction. When such a transaction is initiated, its transaction starts with the same state recorded in the predecessor configuration — ie, the transaction handle's state table is empty. During the transaction, operations are performed, causing object modifications, object creations, object deletions, and/or the import of objects from other configurations; as operations are performed, the state table records the modifications performed to the state inherited from the predecessor configuration.

If a transaction starts with the null configuration, and yields an initial configuration, it can be called an initial transaction. A design transaction can also start inside another ongoing transaction, in which case it is a subtransaction.

A subtransaction imports objects (actually, their state) from its super-transaction. Often, a subset of the super-transaction's state is imported, since the subtransaction is frequently used to model task decomposition (see Section 5.3). Committing the subtransaction typically updates the state of the super-transaction with the changes made in the subtransaction.

Figure 6.5 shows a successor transaction which has one ongoing subtransaction. The subtransaction is operating on a subset of the super-transaction's state.

6.2.2 State Clashes

State clashes occur when a new configuration to be folded in contains objects which already exist in the current transaction handle, but the versions of the objects to be folded in differ from the versions in the transaction handle. That is, we are attempting to import two different states for the same object. For instance, a given application may depend on a particular version of an interprocess communication library; if it is advantageous to include a new module in the application, which depends on a different version of that library, the two may interfere. In Figure 6.6, object A exists in two configurations; the state of A in one differs from its state in the other. An attempt is made to produce a new configuration which uses both U and V, which

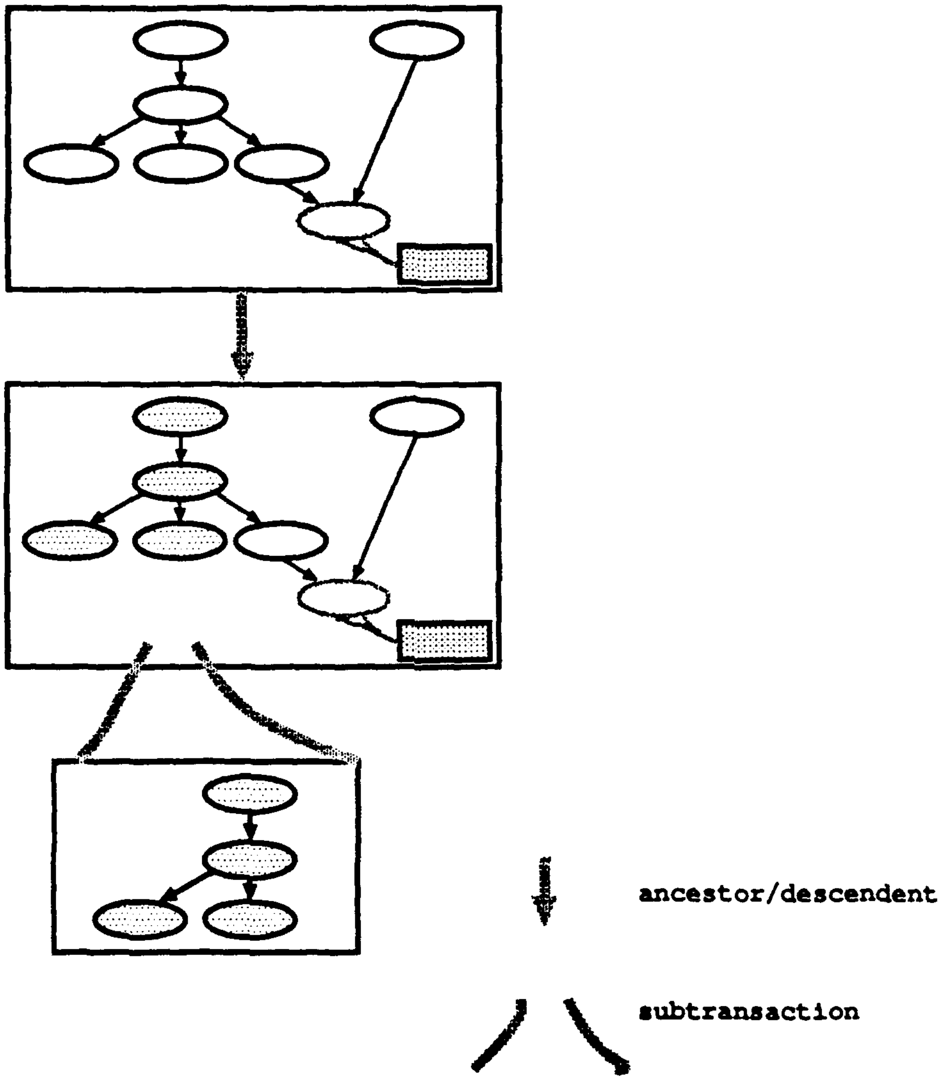

Figure 6.5: A successor transaction and a subtransaction.

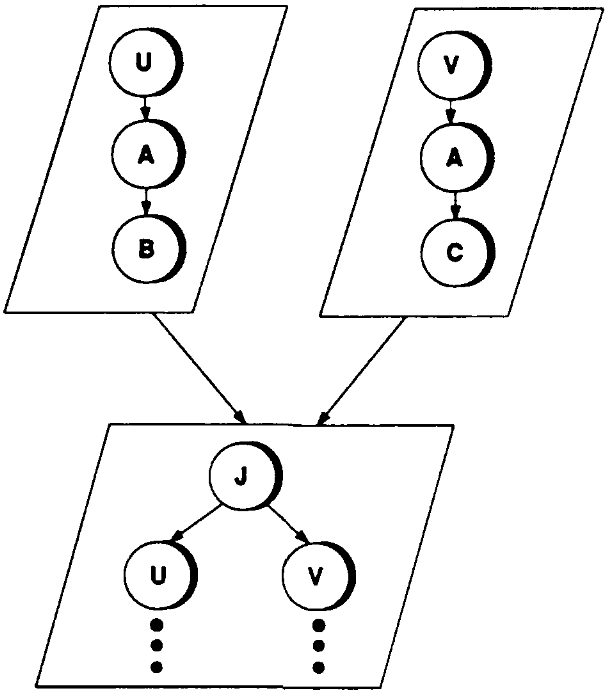

Figure 6.6: A state clash between U's A and V's A.

depend on different states of A.

In general, there are three options in such a situation:

override The incoming state overwrites the current state.

Thus, if U already exists in the current transaction, then when V is imported ":override", then both U and V will reference the state of A which is being imported with V — the state of A which references B.

submit The incoming state is discarded; the current state is maintained. Objects in the incoming state which reference objects with clashing state now refer to the objects as they already exist in the current transaction handle, rather than as they existed in the configuration being imported.

Thus, if U already exists in the current transaction, then when V is imported ":submit", then both U and V will reference the state of A which had been previously imported with U — the state of A which references C.

separate The incoming state occupies a private namespace, as if the incoming objects were duplicated and renamed. This is the effect which occurs when cross-configuration references (version-cursors) are used.

Again, assume U already exists in the current transaction. J's reference to V will be not be a direct reference to a local object, but rather an indirect reference to the V in the imported configuration.

6.2.3 Operations on Transaction Handles

The following are some candidate user-visible operations which might be defined for transaction handles:

```
(make-transaction
     [ :successor-to <configuration> |
          :subtransaction-of <transaction>
              [ :mutable-super | :immutable-super ] ]
     [ :exporting <object>+ ]
     [ :satisfying <change-request>+ ]
 )
```

- Creates and returns a transaction handle object (the holder of a transaction handle object can view and modify objects under change control).
- The new transaction may be a :successor to a configuration (ie, to a committed transaction), or it may be it may be a :subtransaction of another transaction.
- One can specify which objects will be :exported by this configuration as its primary externally-visible objects.
- One can specify one or more change requests (which might be change notifications) which this configuration is intended to satisfy.
- Make-transaction will warn if the predecessor or parent transaction already has an ongoing successor transaction or subtransaction.
- For a subtransaction, one can specify whether the state of the super-transaction should be seen as :mutable or :immutable. If the super-transaction is mutable, any changes made to it during the subtransaction are visible within the subtransaction.

```
(commit transaction :change-descrip <some-class>
                    [ :exporting <object>+ ]
                    [ :compatibility-attributes <some-class> ]
                    [ :no-update-super ]
 )
```

- Commits a transaction, making it read-only and globally visible — ie, making the transaction handle a configuration. One must own the transaction handle to do this.
- Committing a transaction asserts that the objects and state therein are internally consistent and are consistent with each other. The human doing the commit is asserting that the primary representations of each object are consistent with each other; the commit operation runs class- and application-specific verification methods, if any, particularly on the secondary representations.
- One should specify a change description which documents the transaction.
- One can specify which objects will be `:exported` by this configuration as external objects.
- One can specify compatibility attributes which identifies other configurations with which this configuration is compatible.
- By default, Committing a subtransaction causes its objects to be written into the super-transaction (ie, a

  ```
  (load-configuration <this-subtransaction>
  :clash-resolution 'override)
  ```

  occurs). The `:no-update-super` keyword suppresses this behavior.
- Configurations and transactions who have version cursors referencing the predecessor configuration may receive change notifications, if their configuration schema so indicate.
- Commit will warn if there are uncommitted subtransactions.

`(abort transaction [ :discard ] )`

- Terminates transaction without attaching the transaction handle to the configuration history DAG as a successor transaction.
- Discards the state of this transaction if `:discard` is present.
- One must be owner of transaction to do abort.

`(in-transaction T)`

- Makes the transaction of handle T the current transaction for this session, giving access to the objects and state defined in T. If another transaction was previously current, the state defined in the previous transaction becomes inaccessible.
- One must have read/write access to the transaction handle to do in-transaction.

```
(load-configuration
      <configuration>
      [ :clash-resolution interactive|override|submit|separate ] )
```

- Loads the objects and state from the named configuration into the current transaction's proto-configuration.
- If an object being loaded already exists in the current transaction, then the states of the two versions of the objects are checked. If the states are not identical, then a user-specified conflict-resolution strategy is employed:

  interactive Query the user per clashing object.

  override The objects being read in override the current state.

  submit The objects being read in are overridden by the current state.

  separate The objects are not copied in; rather, references to those objects are version cursors, cross-configuration references to the objects in `<configuration>`. They continue to occupy a separate state-space, as if they were duplicated and renamed.

```
(select-objects [ <configuration>|<transaction-handle> ] <object>+)
```

- Returns a proto-configuration object containing only the objects listed, and the objects referenced by those objects transitively.
- Typically used with make-transaction or load-configuration, where some or all of the objects (and state) in the predecessor / super- / to-be-loaded transaction are "copied" into the current transaction.
- With select-objects, one can select which objects should actually be "copied" from the predecessor or super-transaction (objects referenced by the selected objects are also "copied").

```
(find-configuration [ :exporting <object>+ ]
                    [ :selection-rule <some-class>+ ] )
```

- Returns a configuration object.
- One again has the option of specifying one or more objects which the configuration must export.
- One can specify one or more selections rules (based on the compatibility attributes recorded with the configurations.)
- Typically used as the argument of a load-configuration.

There are a variety of other operations which might be useful, including:

- Backing out a commit.
- Re-parenting a subtransaction.

6.3 Configuration Schema

> It is useful to be able to specify a configuration abstractly, including component references which are not resolved until configuration-construction time. A *configuration schema* specifies how to build (or recognize) a consistent configuration, and how to correctly propagate change notifications. A configuration schema specifies: the structure of the configuration (including rules for identifying objects to be fetched from other configurations); how to verify the consistency of the configuration; how to construct the configuration; and how to control the propagation of change within and between configurations.

A *configuration schema* is an abstract specification of a configuration, the schema for a consistent system of objects; a configuration is instantiated from a configuration schema. The configuration schema is a set of rules specifying how to build — or how to recognize — the right set of objects, constructed from the right components and utilities.

The configuration schema is part of the configuration — different versions of it may exist in different versions of the configuration. But it is intended that the configuration schema will change more slowly than the configuration itself; it should be more abstract, describing more than one version of the configuration. Further, when the design transaction is committed, all references in its state table become fixed references; however, the configuration schema remains an abstract specification, including the version cursors.

We have elected to consider the abstract structure of a configuration as a separate specification from the objects making up the configuration. In many systems, dynamic references are embedded in the configuration itself — placed directly into objects' slots, and thus sprinkled throughout the system of objects. We choose to abstract out the dynamic parts of the configuration — dynamic references, verification description, build description, *etc.*— and specify them separately, in a configuration schema.

Typically, a configuration schema will be developed and elaborated over a period of several transactions. It summarizes experience regarding how to properly construct, and properly verify, versions of this configuration. It thus abstracts the activity of a sequence of transactions; it is an abstraction of the "transaction audit trail", replay information. Therefore, the configuration schema specifies not only the st⁻ucture of the configuration, but also the process to be followed in producing or verifying it.

A co̲nfiguration schema includes:

Structure description The abstract description of the structure of the configuration identifies the components and utilities required, and how their interfaces are composed. Objects may be specified by cross-configuration references, including dynamic references; resolution of a dynamic reference may depend on the current environment and the desired operation. For example, in Figure 6.8, npasswd references one of two possible be's; be_{fs} (filesystem-based back end) or be_{yp} (daemon-based back end) . Selection of the appropriate be can depend on the environment for which npasswd is being built. The structure description should specify non-explicit dependencies; structure which is inherent in the objects and their references, and which does not involve dynamic references, need not be specified.

Interface description The description of the configuration's interface identifies the objects which the configuration *needs* from other configurations (ie, the version cursors); it also identifies the objects which the configuration *provides* to other configurations (ie, the objects which it is believed should be the external objects, that other configurations may choose to reference).

Verification description The description of how to evaluate the consistency of the configuration may include several things:

- The dependencies between objects, including the dependencies of derived representations (*e.g.,* A.exe depends on A.lisp).
- Regression tests, including the use of V&V tools.
- User-specified constraints on the attributes of objects.

Process description The description of how to construct the configuration (ie, to establish consistency) tells how to generate or update dependent objects (including derived representations), and in general how to construct and compose objects which must be constructed or composed. These "actions lines" may be conditionalized on the current environment and the desired operation. For example, in the VMS environment oms.ada would be compiled with a VMS compiler (and a particular system.ada), but if the configuration is being build in the Unix environment, a different compiler and a different system.a would be used. What part of the process description is actually executed at consistency-establishment depends on the areas of inconsistency identified via the verification description.

Change propagation The description of how to propagate change into this configuration from foreign configurations may simply be the interface description's dynamic references. For example, if one of the dynamic version cursors had previously resolved to C:1.9:, but the newly-created C:2.0: would also satisfy it, then by default a change notification message should be sent to this configuration's owner. More generally, it should be possible to specify what action, if any, should be taken in response to a dynamic reference becoming unsatisfied or resatisfied (see Section 6.3.1).

A "Makefile" can be considered a weak configuration schema. In its structure description, the only relevant property of the objects are their relative timestamps. No interface description is provided. The action lines of the build rules form the process description; the operations specified there cannot be conditionalized. The only inherent verification description is the build rules, which define consistency based on relative timestamps. Cross-configuration change propagation is not defined, though intra-configuration change propagation is handled by the build rules.

Our understanding of the requirements which must be satisfied by a notation for configuration schemas (and for version cursors) is fairly detailed; we have not chosen a particular notation sߦ far. Further work must be done hypothesizing and testing notations.

6.3.1 Change Propagation and Change Notification

There are several benefits which arise from our model of configurations and transactions.

Change propagation and deep copy. Because we do not model change by copying individual objects, we avoid the need to do change propagation for check-out operations (Section 6.1).

Disambiguating the propagation path. When dependencies among objects form a DAG rather than simply a tree, then there are multiple possible paths by which change can propagate through a system. If all paths are followed, a proliferation of uninteresting or unintended versions occurs. A mechanism for group check-in/check-out allows the dependencies to be disambiguated ([15]). Our transaction serves as such a delta set (one logical change subsuming multiple physical changes). Thus, other configurations and transactions need only to react to the entire transaction, not to the various intermediate operations within it.

Cross-configuration vs Intra-configuration propagation. We can distinguish between cross-configuration change propagation and *inter-configuration* change propagation, and specify different strategies for each.

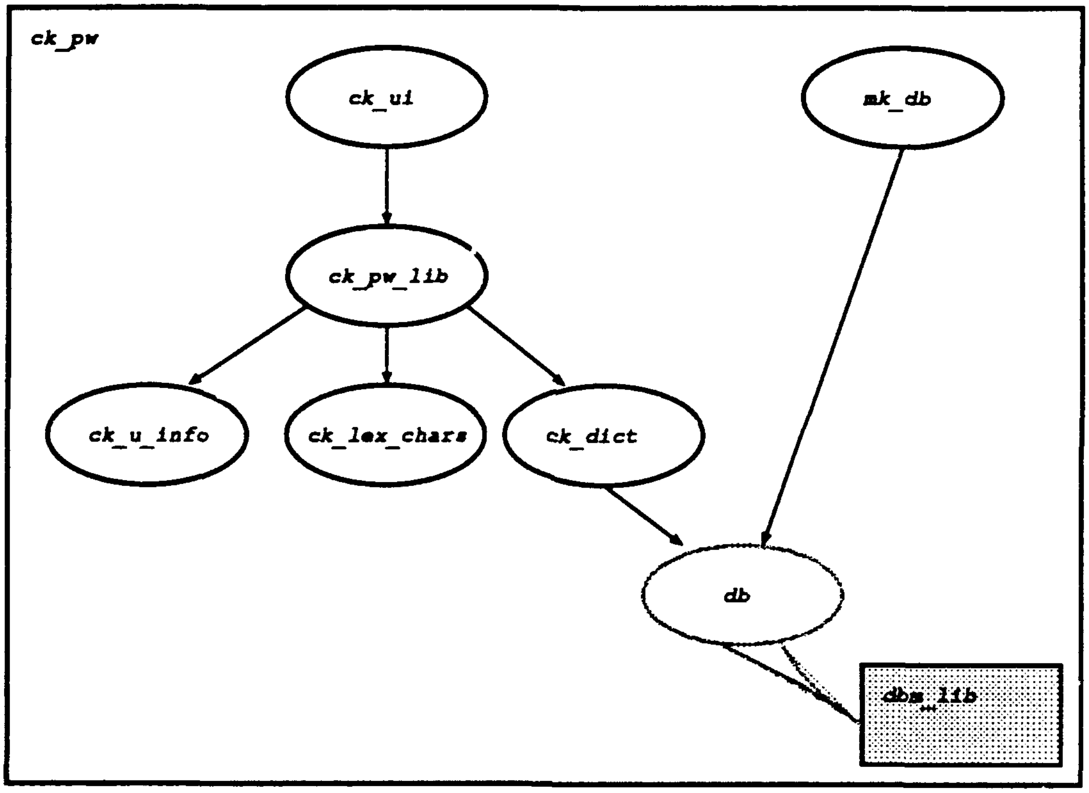

Figure 6.7: Second version of Check-Password-Quality configuration.

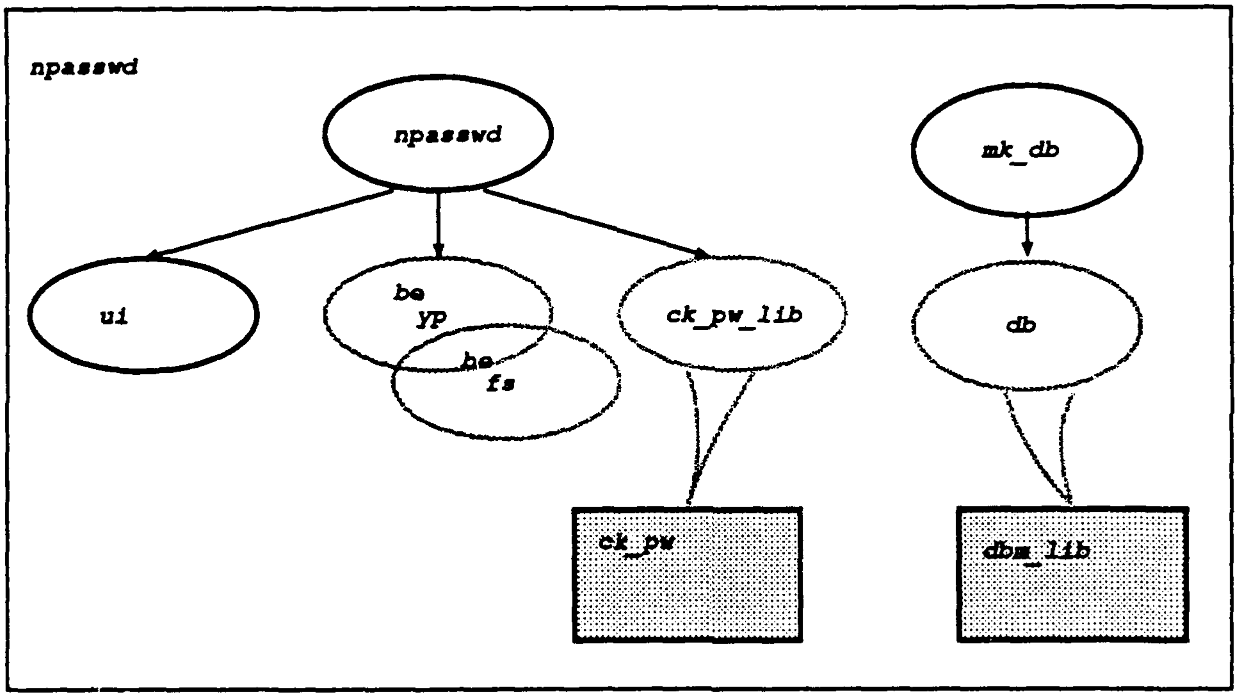

Figure 6.8: Password-Changer configuration, descended from Check-Password-Quality

Configuration schema may be used to control cross-configuration change propagation. For example, if one of the dynamic version cursors had previously resolved to C:1.9:, but the new C:2.0: would also satisfy it, then by default a change notification message should be sent to this configuration's owner. More generally, it should be possible to specify what action, if any, should be taken in response to a dynamic reference becoming unsatisfied or resatisfied. It should be possible to specify the types of changes which should cause a dynamic reference to send a change notification, the types of changes which should be ignored, and the types of changes which should cause action to be taken automatically (*e.g.*, a rebuild).

Intra-configuration change can typically be handled by a passive (flag-based) strategy, augmented with user-defined constraints. For many changes, we can postpone resolving the inconsistencies arising from such changes until consistency-establishment or configuration-construction occurs. For some changes, we will want dependent objects to react immediately; user-defined constraints allow such needs to be specified.

6.4 Cross-configuration and Dynamic References

> A configuration schema may include references to objects which will be fetched from other configurations — cross-configuration references, or *version cursors*. The references may be dynamic — ie, the particular source configuration may not be chosen until configuration-construction time. A dynamic version cursor includes (a) the object signature and (b) a rule for selecting an appropriate version of that object (ie, an appropriate source configuration). Dynamic version cursors provide for flexibility in evolving configurations, and allow references to hypothetical objects which will be constructed in other transactions.

In the configuration schema, when describing a component, we may specify that it should be imported from a foreign configuration (ie, it is not or may not yet be in the current transaction). A reference to a component in a foreign configuration — a cross-configuration reference — is a *version cursor*. The version cursor may be static or dynamic. Since it must resolve to a particular version of an object, it must specify:

The object's signature. This may be a name, an interface or protocol specification, or something more complex.

A version-selection rule. This may be "version :1.5:" (for a static cross-configuration reference), or a more-or-less complex dynamic reference — *e.g.*, ":binary compatible with currently loaded windowing system:" The form of version-selection rules depend heavily on the form of compatibility attributes (see Section 6.1.2).

In many systems, cross-configuration references are sprinkled throughout the system of objects. We choose to place version cursors in the configuration schema; in the system of objects, dynamic references show up as *hypothetical objects*, which are created when a dynamically-specified configuration is loaded. We believe that this centralization of dynamic references will reduce the frequency of the situation where different parts of a configuration reference antagonistic versions of the same foreign configuration.

There are a variety of characteristics which could be included in a version cursor's version-selection rule:

A static configuration identifier The selection rule identifies a concrete configuration ("C:2.1:") when the version cursor is a static reference.

Compatibility requirement We may specify that we want a version which is compatible with some other configuration. The other configuration may be a known concrete configuration, or it may itself be a dynamically-chosen configuration. In the latter case, backtracking may be necessary to discover a consistent (or the "best" consistent) system of configurations.

Change request responses We may specify that we want a version which has responded to particular change requests (ie, which has fixed certain deficiencies).

Environment and operation dependencies We may conditionalize the request on the environment or desired operation, particularly for configurations which are intended to be constructed for multiple environments or to satisfy multiple purposes.

Dynamic version cursors make significant use of compatibility attributes in specifying and resolving version-selection rules. We have described compatibility attributes as being at least labeled directed arcs (Section 6.1.2), which specify that one configuration can satisfy a request asking for another configuration, in the context of a particular environment and a particular operation. We are assuming the version-selection rules are first-order logic expressions. The selection rules and the version-search process must be adequately powerful; but if the expressions become too general, then the computational complexity of producing a configuration may become inordinately high, and understanding how partial resolution of a configuration schema affects the rules may be difficult. We currently believe that limiting the expressions to Horn clauses which depend on compatibility and transformation information is acceptable, but this needs to be validated.

6.5 Initial Experience using the Model for Avionics CAD

An in-house project is implementing part of the KBSA framework CCM model to solve CCM problems in a prototype avionics design capture system.

In their avionics CAD domain, multiple teams of designers, responsible for different subdomains, are evolving large and complex subsystem designs which must be periodically integrated to yield a design of the complete system. A key requirement is to have an on-line model of the system's baseline configuration, and of any options which may be instantiated to construct particular concrete configurations; it must be possible to determine the consequences of the choices which were made in constructing a particular concrete configuration.

A typically development process, in the abstract, would look like this.

- The system administrator creates the database and defines the aircraft class which the database describes.
- The system integrators for the various subdomains define the subsystems from which data is required. Working with the system administrator, they also define user rights for the various subsystems.
- The designers within the various domains create and elaborate the design objects making up their subsystems, including intra- and inter-subdomain interfaces.
- The system integrators make sure that the data entered is consistent, complete, and on schedule for system integration releases. The system integrators also assist in any coordination among teams that may be required.
- For many design objects, some features of the object will never change in future revisions, but other features will be subject to frequent change. Depending on the subdomain, revision management is the responsibility either of the subsystem designer or the subdomain's system integrator.
- A distinguished system integrator is responsible for aircraft configuration management (ie, for configuration management of the complete system).

6.5.1 Representing Change

It was necessary to retain the revision history of these avionics parts objects — for example, the descriptions of fielded revisions must be available as long as the revisions are in service. Saving the entire database — a snapshot of the world — each time a configuration needed to be retained would require approximately 300 gigabytes per class of aircraft (approximately 100 subsystems each requiring about 30 snapshots); this was deemed undesirable.

The first-cut solution was to maintain a revision history on an object-by-object basis; an object was checked-in by marking it "immutable", and a configuration was simply a list of pointers to frozen (checked-in) objects. Any future changes to a frozen object would need to be made to a new version of that object — ie, to a mutable copy of the object, created by a check-out operation. But the database is highly interconnected; a change to one part of the database has effects beyond the strictly local modifications. When checking-in or checking-out an object, the objects which it references and which reference it must also be dealt with (see Section 6.1). When checking-in an object, it is necessary to also check-in the objects which are components of that object, since they are part of its definition; when checking-out an object, its components must be checked-out also. The more difficult question was how the objects which reference an object should respond to a new version of that object. Copy propagation problems arise; the objects which reference the old version of the object may need to themselves be duplicated, since a new version of the object referenced is a change to a feature of the objects which reference it. Thus it was necessary to produce not only a new version (copy) of the object to be checked-out, but also potentially of all the objects which referenced it — in the worst case, copying the entire database again.

Criteria which could be used to limit the recursive copy propagation without sacrificing correctness and consistency were not readily apparent; we contend that this will be true of most domains. The KBSA CCM model, which is conceptually clean and leads to a space-efficient representation of a tree of configurations, was therefore chosen.

A requirement for this avionics domain was to facilitate the work of multiple cooperating teams via decoupling the subdomains during development, while facilitating system integration. The multiple subdomains were decoupled into multiple trees of configurations. Each subdomain maintains their configurations as a tree of state tables, where each state table records the modifications made to the database since the previous state table (see Figure 6.9). Configurations from the subdomains are periodically composed to form one global system configuration, a node in the system configuration tree; the system configurations cache the complete state of the system.

```
Read-Release (release-name)      ; read from file
    RETURN: release
Write-Release (release-instance) ; write to file

DefDomainRelease (domain new-release-name predecessor-name)
    RETURN: release
DefGlobalRelease (new-release-name predecessor-name)
    RETURN: release

Create-object (release class-name &rest slot-value-pairs)
    RETURN: hash-id
    SIDE-EFFECT: Check that release isn't frozen.
                 Check slots for completeness, consistency.
                 Create instance of object and insert it into release.

Access-slot-value (release hash-id slot-name)
    RETURN: slot-value
    SIDE-EFFECT: Traverses parent links of releases until slot value is
                 found.
                 Check that object has not been deleted.
Replace-slot-value (release hash-id slot-name slot-value)
    RETURN: slot-value
    SIDE-EFFECT: Check that release is not frozen.
                 Check that slot is consistent, user-setable.
                 Unless an instance of object is already present in release,
                     Create an instance of object and insert into
                     release, copying slot-values from parent releases.
                 Change existing value of the slot to slot-value in release.

Query (release &rest slot-value-pairs)
    RETURN: (hash-id*)
    SIDE-EFFECT: Calls Access-Slot-Value to find objects in database
                 that match the slot-value-pairs description.
```

Figure 6.9: Configuration Management Operations in Avionics Application.

6.5.2 Dynamic References and Change Notification

As mentioned, a requirement for this avionics domain was to facilitate the work of multiple cooperating teams via decoupling the subdomains during development, while facilitating system integration. The multiple The use of dynamic version cursors supports controlled inter-domain references during development, facilitating system integration. If an object in subdomain X must reference an object in subdomain Y, the object's definition is not referenced directly; instead of a static reference, a dynamic reference is used — a query statement (dynamic version cursor) which may be "satisfied" by zero or more objects in the Y subdomain.

list of unsatisfied queries is maintained, and the list must be cleared before a consistent release can be captured; this is the "system integration" (consistency establishment) process. The distinguished system integrator is responsible for deciding the cause of unsatisfied queries; that individual may direct the submitting design group to remove the query, direct the submitting group to restate the query, or direct the target design group to satisfy the query.

An interdomain communication protocol for queries was defined; database actions/transactions observe the protocol to ensure database consistency. Creation of a version cursor initiates the query protocol, which is used to inform one domain (the donor domain, which receives the query) that an object in another domain (the recipient domain, which sends the query) wants to depend on or "point to" an object in the donor domain. The query protocol guarantees that the interdomain links and back-links between objects are set only upon explicit agreement from both domains.

A slight complexity is introduced when the target of a previously satisfied query is changed, resulting in a now-unsatisfied query. The dynamic version cursor capability also supports a change notification facility which notifies a user of an object if the object has been changed such that it no longer satisfies the version cursor's selection rule.

Another interdomain communication protocol was defined, for change notifications; change notifications inform one domain (the recipient, which receives the notification) that an object in another domain (the donor, which sends the notification) has been modified so significantly that the links and back-links between objects in the recipient domain and the modified object in the donor domain should be reevaluated.

The protocols for query satisfaction and change notification are computer assisted, requiring the okay of system integrators from both domains.

The prototype avionics design capture system has been implemented in Common LISP, and tested with small test problems, it is now being exercised with real data

(9000 objects, about 10 megabytes of avionics design data).

6.6 Related Issues and Open Issues

There are a variety of issues tangentially related to CCM which the KBSA framework CCM :nodel does not yet address; the model also requires exercising and improvement.

- *Exercising the Model.* The CCM model has a solid conceptual base, but some key areas require further elaboration, including:
  - How should environments be described?
  - What operations on design transactions should a working system provide?
  - Can multiple representations for the same conceptual object be tightly associated with each other, such that they can be referenced and managed in a conceptually clean fashion?
  - How should "back-transformations" — transformation which modify the source representation as well as the target representation — be managed?
  - What are good notations for compatibility attributes, version-selection rules, and configuration schema?
  - What form should compatibility attributes take? What are the precedence relations? Is it possible to determine default compatibility attributes?
  - How should version-selection rules and the version search process be structured to provide sufficient power without inordinate computational complexity and without sacrificing comprehensibility?
  - How should CCM be performed on a distributed object-base? Timestamp-based distributed transaction management bears striking resemblance to CCM; can that similarity be leveraged?
  - How can the process record summarized in the configuration schema, and the experience record summarized in the compatibility attributes, be leveraged in an integral design documentation / history mechanism?
  - How should the configuration schema interact with subtransactions? Can it serve to define the contracts which the subtransactions should satisfy?
  - Can more assistance be provided for coping with concurrent and overlapping changes performed to the same configuration or system of configurations by different teams?
  - How can the change propagation control specified in version cursors and configuration schema be used to do automated impact analysis for hypothesized changes?

- How can the history and status of successor transactions and subtransactions be summarized to display the status of an ongoing change, and to estimate time-to-complete?

The model is currently analogous to a set of axioms and some core theorems — it needs to be prototyped and exercised, to expand the number of theorems and to evaluate a..d improve the model.

- *CCM and Activities Coordination.* It is clear the CCM and the activities coordinator will be closely related. Both are concerned with managing the changes which occur to project state, and with facilitating (or automatically performing) some operations on project state while prohibiting others.

  The activities coordinator will undoubtedly place further restrictions on the legality of operations (based on the role of the person involved) in addition to access control restrictions; it will play the major role in enforcing *process consistency*, to better lead to the *product consistency* which is the major concern of CCM. In addition, much of the compatibility information will be usable by the activities coordinator to determine legality of a particular operation. We believe that the determining factor for what capabilities go under the auspices of CCM and which go under the activities coordinator should be based on the notion of consistency; CCM is trying to deal with the consistency of the product, whereas the activities coordinator is trying to deal with the consistency of the process.

  The activities coordinator will be a key resource in automating the CCM process, to reduce its intrusiveness as much as possible, and to extend its benefits to the full scope of both informal and formal changes. The activities coordinator will also play a key role in proving configurable CCM policy; an integrated CCM facility should provide the *mechanism* necessary to implement the *policy* chosen by the project.

- *Knowledge-Based applications of CCM.* It would be possible to use CCM to model "possible worlds" from knowledge-based tools or bindings from a unification (logic-based) system. Although we did not pursue the issue in depth, we identified two issues that interacted with our CCM model, namely granularity and truth maintenance. How finely grained should the changes be which are supported in a CCM model — and, correspondingly, how many transactions per unit time will be involved? We believe CCM will typically manage significantly coarser-grained objects than would be needed to model knowledge-based or unification-based systems. The purpose of most TMSs (Truth Maintenance System) is to record decision dependencies and to reason about changes in them;

in CCM, we do not know how to record (and use) information about why development on a particular alternative configuration was halted, or why a user decided *not* to use a particular configuration.

- *Reuse.* Combined with better interface specifications, the KBSA CCM model has the ability to record compatibility information in a machine manipulable fashion. Just as early software developers advanced from having textual descriptions of how to assemble systems, to "scripts", to "Makefiles" that have a specific language to reason about the construction of systems, we believe that making compatibility information explicit will make it possible to determine if some existing piece of software will work in one's environment. This is a critical part of the software reuse problem — namely, having confidence that expending the effort to use the software will be successful. While CCM modeling will be able to determine the compatibility of a software module and make integration of the component easier, the problem of identifying what components may be applicable i<span>s</span> outside of the scope the CCM model; the CCM model deals with issues of compatibility and recoverability, but only marginally with issues of module classification and library search.

Part III

KBSA User Interface Environment (KUIE)

Chapter 7

Introduction to KUIE

As part of the Knowledge Based Software Assistant (KBSA) Framework [1] project, Honeywell has developed the KBSA User Interface Environment (KUIE). KUIE applies Object-oriented design, specification, and prototyping tools, techniques and methodologies to the task of constructing graphical, direct manipulation, interactive user interfaces. The environment is highly Object-oriented and is based on the Common Lisp Object System (CLOS) [17, 18].

The overall design goal of the KUIE development effort was to provide a specification driven, window based, interactive, user interface construction toolkit which removes much or all of the burden of interface construction from the application programmer.

Several of the characteristics of the KUIE design were drawn principally from Texas Instrument's Common Lisp User Environment (CLUE) [8]. The design also was influenced by the Knowledge Based Requirements Assistant (KBRA) presentation mechanism [19] and other object oriented user interface systems. KUIE expands these systems by utilizing the extensive leverage provided by the advanced object oriented facilities provided in CLOS. This has simplified KUIE's interface and increased its flexibility.

In addition to maintaining a strong object oriented flavor in KUIE and compatibility with CLOS, we avoided dependence on any single hardware or product platform by embracing the *X Window System* [6] (hereinafter referred to as *X*) as the implementation base. *X* (and its Common Lisp interface, CLX [20]) is powerful, widely available, and rapidly becoming the *de facto* standard for windowing systems in the workstation, minicomputer, mainframe, and supercomputer markets.

Building the user interface portion of contemporary interactive systems is a complex and demanding job. A frequently stated rule of thumb is that user interface design and implementation typically consumes one-third of the total system development effort for large systems. Even more disturbing is the fact that little, if any, of the user interface subsystem design or code developed by one project is ever re-used in another: the vast majority of user interface subsystems are built from scratch.

This sorry state of affairs stems from the vast numbers of minute details which must be mastered when designing modern user interfaces and the impact they have on the rest of the system's design and implementation choices. Although modern software engineering "best practices" dictate that the user-interface subsystem be carefully separated from the remainder of the system to be constructed, this often is difficult to achieve with acceptable results.

Recently, tremendous progress has been made in the development of low-level user interface construction building blocks; chief among these is *X*, originally developed by MIT and now supported by the *X Consortium*.[1] *X* provides a low-level system upon which sophisticated user interfaces can be built for modern execution/presentation environments, *i.e.*, a large, high resolution, bitmapped display, keyboard, pointing device (*e.g.*, mouse), and plenty of raw processing power (*e.g.*, 3–20 MIPS) and memory (*e.g.*, 16+ MB).

Even though the support provided by *X* is at a relatively low level, it effectively isolates applications from dependencies upon the input/output devices. *X* also provides the unique capability to isolate application programs from intervening networks. Its client-server mode of operation allows the application program to execute on one machine (*e.g.*, a supercomputer at a central site), with the application "window" appearing on the display of another (*e.g.*, the user's workstation display down the hall or across the country), all transparently to the application program.

The basic facilities provided by *X* are too primitive for direct use in all but the simplest applications. Thus, most existing applications make use of one of the various *X Toolkits*. These toolkits provide support for the now familiar features of bitmapped graphics interfaces: pop-up/pull-down menus, scrollbars, sliders, control panels, push/radio buttons, title bars, *etc.* These facilities greatly simplify the construction of user interfaces and improve their quality, but they do not relieve the application programmer of all (or even most) of the many details involved in constructing the application interface. An example of this type of toolkit is TI's CLIO [21], which is built upon, and intended to be a companion of, CLUE.

Looking at this from a slightly different perspective, present applications built upon *X* are relieved from the responsibility of managing the details of the periphery of

[1] X Consortium, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.

the application (*i.e.*, what happens *around* the application window), but they are still totally responsible for what happens *within* it. There are no support facilities available which help the application programmer handle redisplay, pointer tracking, "rubberbanding", highlighting, moving, resizing, or placing displayed objects. The application programmer must be intimately aware of the details of *X* event handling and is totally responsible for managing the appearance of the application's "windows".

KUIE can be described as an attempt to provide a different kind of *X Toolkit*: KUIE automatically manages the appearance of the application's windows, based upon ***declarative*** (as opposed to ***procedural***) guidance provided by the application programmer, and provides the application programmer with the means to accomplish easily many commonly used interface paradigms (*e.g.*, rubberbanding, highlighting, mouse sensitivity) without becoming involved in the intricate details of event handling and the complex programming this normally entails.

Because KUIE is based on CLOS and provides a (primarily) declarative interface to the application programmer, it is possible to construct sophisticated user interfaces with relative ease (compared to the complex programming task previously required). Additionally, since a "KUIE interface" will be primarily a list of CLOS declarations, as opposed to thousands of lines of complex functions, reuse of user interfaces (or portions of interfaces) across multiple applications will be more easily achieved.

7.1 Design Goals

The following five principle goals guided the design of KUIE:

Object Oriented Abstractions The benefits of object oriented abstractions for user interfaces have been demonstrated in several operational systems [22, 23]. The approaches taken proved to be both capable of managing the complexity inherent in user interfaces constructed for applications, and yet be highly flexible for constructing novice user interface features [24].

CLOS Compatibility The ratification of CLOS as part of the Common Lisp language [18] has provided an industry-wide base for object oriented programming. Close association with the CLOS standard will aid KUIE's use and acceptance by a larger community.

Portability Applications should be easily portable to any hardware/software environment which supports CLOS and the *X Window System*.

Modularity KUIE should comprise a well-defined and self-sufficient laver of the user interface programming system. Using KUIE, an application prog…mmer should be able to implement most types of user interfaces without accessing underlying software layers and without knowledge of the implementation internals of KUIE objects.

Extensibility The KUIE interface should provide the ability to define new types of user interface objects which refine and extend the behavior of the basic object types. KUIE provides this ability through the CLOS features of class specialization, method mixing, and inheritance.

Basing KUIE on the emerging Common Lisp/CLOS and *X* standards has enabled us to meet these design goals with minimum difficulty.

7.2 Concepts

7.2.1 KUIE Levels

KUIE is logically and physically divided into three separate levels of increasingly powerful functionality. Level 1 has been implemented and reasonably well-tested. At the time of this writing, it is being used by three different projects at Honeywell's Systems and Research Center. Level 2 has been designed and a prototype implementation has been constructed. Level 3 is simply a figment of our imaginations. Each of these levels provides new, powerful user interface construction capabilities not found in systems commonly available today.

The overall structure of KUIE and its relationship to CLX, CLUE and CLIO are shown in figure 7.1.[2] As can be seen from this figure, all three levels of KUIE, as well as CLUE and CLIO, are visible to the application program. The application program's use of CLUE however, must be tailored to fit within the KUIE framework. Application programs will rarely, if ever, call CLX directly.

The three levels of KUIE a‹e:

Level 1 — Building Blocks. This is the lowest level of KUIE. It provides *X Toolkit*like facilities for constructing the basic pieces of a window-based, interactive user interface, while hiding as much of the low-level details of the interactions between the application and the *X Server* as possible.

[2] KUIE Level 3 does not yet exist; the figure shows its planned place in the overall structure.

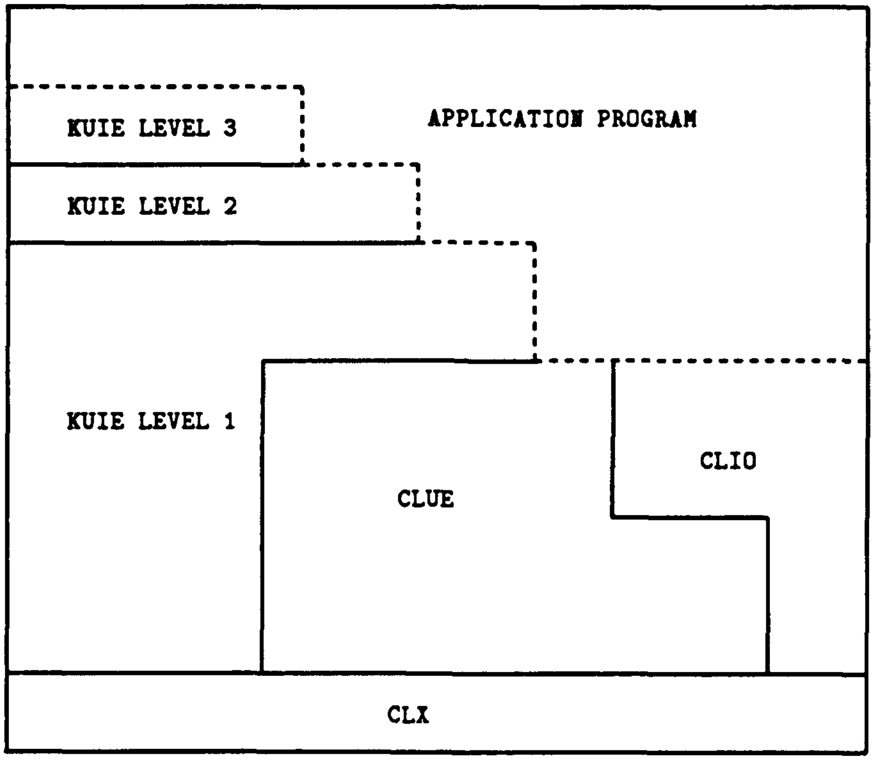

Figure 7.1: Structure of KUIE, CLX, CLUE and CLIO

This level of KUIE is an extension of TI's CLUE, which is itself an extension of CLX, the Common Lisp interface to *X*. This level of KUIE, together with the conventional toolkit facilities provided by CLIO and the basic windowing facilities provided by CLUE/CLX, provides application programmers with a very powerful, object oriented paradigm for constructing interactive user interfaces.

Level 2 — Automated Layout. Below this level, the application programmer is completely responsible for the layout of the objects displayed within the application's window(s). For some applications (primarily text-based), this is not a major concern. However, for more graphics-oriented applications (*e.g.*, the KBRA [19]), the intelligent placement of displayed objects has a major impact upon the understandability of the information presented to the user.

This level of KUIE provides semi- or totally automated layout of the components of a user interface — without the need for tedious and error-prone placement algorithms supplied by the application programmer — based on a simple constraint language/maintenance system which operates upon values of and relationships between the displayed objects.

Level 3 — Gesture Recognition. Below this level, the application programmer has been freed of many of the onerous details of user-interface programming. However, he is still totally responsible for interpreting the meaning of the various actions the user applies to the displayed objects.

The purpose of this level of KUIE is to simplify the application programmer's input processing task by translating the user's manipulations of the graphical interface into direct calls upon the relevant action routines within the application program. The application programmer will specify the mapping between user actions and the application routines to be invoked. The complex, detailed and error-prone task of recognizing the "gestures" made by the user and invoking the intended "semantic" routines will be performed by KUIE.

Although the higher levels are built upon the lower ones, the facilities of the lower levels may be used to construct sophisticated user interfaces without relying upon the services of the higher levels if they are not needed.

Thus, the three levels of KUIE each provide progressively more sophisticated capabilities to the application programmer. Each of these capabilities supports and/or automates a different aspect of graphical user interface programming; aspects which must be explicitly handled — often in laborious detail — by the application programmer using window systems and related toolkits commonly available today.

7.2.2 KUIE Graphics

An interactive application program can be considered to consist of a collection of functions, some of which perform the processing that is essential to the application's purpose (*e.g.*, text editing, knowledge base management, *etc.*). Other functions exist solely for the purpose of communicating with the application's human user. In KUIE, such human interface functions are represented by specializations of the KUIE-defined graphic object class. The graphic class defines a primitive graphical object that is capable of being positioned and displayed on the workstation display. Graphics are responsible for presenting application information to the user on the display, and for informing the application program of input sent by the user via interactive input devices (such as the keyboard and the display pointer). A graphic generally embodies a component of the user interface that knows how to:

- display its contents,
- process input events that are directed to it, and
- report its results (if any) back to the application.

A KUIE graphic provides a relatively high-level abstraction for user interface programming. KUIE relies upon the services of a lower-level subsystem typically referred to as a *window system*. The window system provides interfaces for controlling interactive I/O hardware such as the display, the keyboard, and the display pointer. KUIE manages the interface to the underlying window system, relieving the application programmer of this burden. The application programmer deals in terms of objects physically placed on the user's screen which are subclasses of the KUIE graphic class. The purpose of such an abstraction is twofold:

- To simplify and raise the level of the dialog between the application and the user. A graphic insulates the application programmer from detailed behavior of a user interface component (such as displaying its contents and acquiring its input). As an "agent" of the application, a graphic can directly communicate with the user in terms closer to the application's domain.
- To define a uniform framework within which many different types of user interface objects can be combined. The graphic class raises to a higher level the commonality between a great variety of interface objects -- menus, forms, dials, scroll bars, buttons, dialog boxes, text entry, DAGS, trees, charts, tables, *etc.*

A graphic is intended to be similar to a *window* in the underlying *X Window System*. Graphics have many of the same properties and abilities as *X Windows*. Graphics, however, are *not* windows: the *X Server* has no knowledge of them and cannot manipulate them: KUIE is completely responsible for mapping application program requests directed to KUIE graphics into the appropriate window system calls to achieve the desired effects.

Since graphics are intended to be so similar to windows, why not implement them as such? The answer is very simple: most window systems (including *X*) restrict windows to strictly rectangular shapes; KUIE graphics are more general: they can be rectangles, ellipses, regular polygons, lines, arrows, pixmaps, and hierarchically structured objects such as DAGS, trees, tables, charts, *etc.*

Nevertheless, graphics exhibit many of the characteristics of *X Windows*: they may have ***backgrounds*** which are ***transparent*** or ***opaque***; they may have ***borders***; both backgrounds and borders may be ***solid*** or ***tiled***, and may be derived either from a single *color* or from a *pixmap*; they may be ***moved*** and ***resized***; they may be hierarchically organized in a ***stacking order*** relative to their siblings; at any given time, they may be completely visible (***presented***), partially visible (***occluded***), or invisible (***withdrawn***); *etc.*

The subclassing and inheritance properties of CLOS are important to the use of graphics. A graphic subclass implements a specific interface technique for input and output. Thus, a graphic subclass can represent either an extension of a technique, or it can provide a variation in style. This protocol is expected to lead to the development of graphic "libraries", providing a rich repertoire of interface techniques and a choice of several functionally interchangeable styles.

7.3 Constructing User Interfaces With KUIE

KUIE distinguishes two different aspects of programming a user interface:

- using existing graphics, and
- defining a new graphic subclass.

Thus, there are two categories of user interface builders: the application programmer and the graphic programmer. This distinction contributes to the separation of application programming from user interface programming — which is a primary goal of user interface management systems.

7.3.1 Application Programming

The application programmer has knowledge of the application that is being constructed and defines how the screen should appear to the user of the application. He/she will use and combine the existing KUIE graphics to achieve the desired user interface. The application programmer, who instantiates and uses a graphic object, does not need to know how the class and methods of the graphic were implemented by the graphic programmer. In particular, the window system interfaces used by the graphic programmer need not be visible to the KUIE application programmer.

Complete documentation for application programmers for KUIE Level 1 is provided in the KUIE Reference Manual [7]. The application programmer interface to KUIE Level 2 is not yet documented.

7.3.2 Graphic Programming

The graphic programmer constructs new graphics and specializations of existing graphics that may be used in new applications. The needs of the graphic programmer are different than those of the application programmer in that he must be aware of the specific KUIE- and/or CLUE-defined protocols that must be maintained for graphics and how to use the inheritance hierarchy to achieve the desired results. New graphics that require complex screen management operations may require the graphic programmer to use underlying *X Window System* calls. This difference in perspective between graphic programmers and application programmers separates knowledge of the application from knowledge of the user interface.

The KUIE graphic programmer interface is not yet documented.

Chapter 8

Level 1 — Building Blocks

KUIE Level 1 consists of a set of predefined object classes, methods, and protocols that may be used to construct interactive application user interfaces for modern computing environments (*i.e.*, high-performance workstations with bitmapped display, pointing device, keyboard). The KUIE classes define objects which represent graphical images on the workstation display; the methods define the mechanisms by which the graphical images may be manipulated by the application program; and, the protocols define the conventions which must be adhered to in order to extend the *set of* graphical object classes.

The following subsections describe some of the major features of Level 1. For a complete description, see the KUIE Refe.ence Manual ([7]).

8.1 Classes

The KUIE object classes can be divided (roughly) into three groups: graphics, graphic-composites, and miscellaneous classes. The graphic class (and its subclass specializations) define common graphical images: arcs[1], polygons, regular-polygons, lines (which may be multi-segmented and may have arrows), graphic-pixmaps (bitmap images), ellipses, circles, text-boxes, *etc.* All these subclasses of graphic are collectively referred to as "primitive" graphics, to distinguish them from the graphic-composites.

There is also one rather special subclass of graphic: rectangle. Contrary to intuition, KUIE rectangles are not a subclass of polygon. Rectangles are a concession to the restrictions of the *X Window System*. For example, *X* cannot display text or pixmaps

[1] Arcs are not implemented yet.

which are rotated with respect to the *X Window System* coordinate plane. Thus, both text-boxes and graphic-pixmaps are subclasses of rectangle.

The KUIE graphic-composite class defines a class which allows the hierarchical composition of graphics. Although graphic-composites are graphics, it is occasionally convenient to talk about them separately from the primitive graphics. Graphic-composites are a subclass of rectangle and also of graphic-container, which is described in section 8.3.

The miscellaneous classes include various supporting classes which may be used internally to define externally-visible KUIE classes, mix-in classes, and the canvas class. The canvas class defines a specialized *X Window* object upon which graphics (both primitive and composite) may be *presented* (*i.e.*, drawn). It is important to note that graphics may not be displayed upon arbitrary X *windows;* only KUIE canvases supply the required support for presenting KUIE graphics; thus, every KUIE-based application will use at least one canvas object.

The KUIE class hierarchy is shown in figure 8.1.[2] The classes basic-contact, contact, and composite are defined by CLUE. KUIE application programmers will be concerned primarily with the graphic class and its subclasses, although they must be aware of canvas objects. In addition to these classes, KUIE graphic programmers also will deal with the graphic-container class, which encapsulates the functionality required for an object to properly manage subsidiary objects which are graphics.

The KUIE graphic objects created by an application automatically perform many of the low-level interactions necessary to support interactive windowing interfaces:

- they are able to display themselves on the user's screen;
- they are aware of their relationship to other objects being displayed — for example, they understand the concept of "stacking order" and thus, know when they must redisplay themselves in response to changes in the stacking order and/or the geometry or position of other graphics objects;
- they are able to detect when the display pointer is positioned over them and when it is not;
- they are able to detect when keyboard input is directed at them and when it is not;[3]
- they know when they are visible on the display (*presented)* and when they are not (*withdrawn)*.

[2] Object classes shown with dashed boxes are planned, but not yet implemented.

[3] Detecting the keyboard focus is not yet implemented.

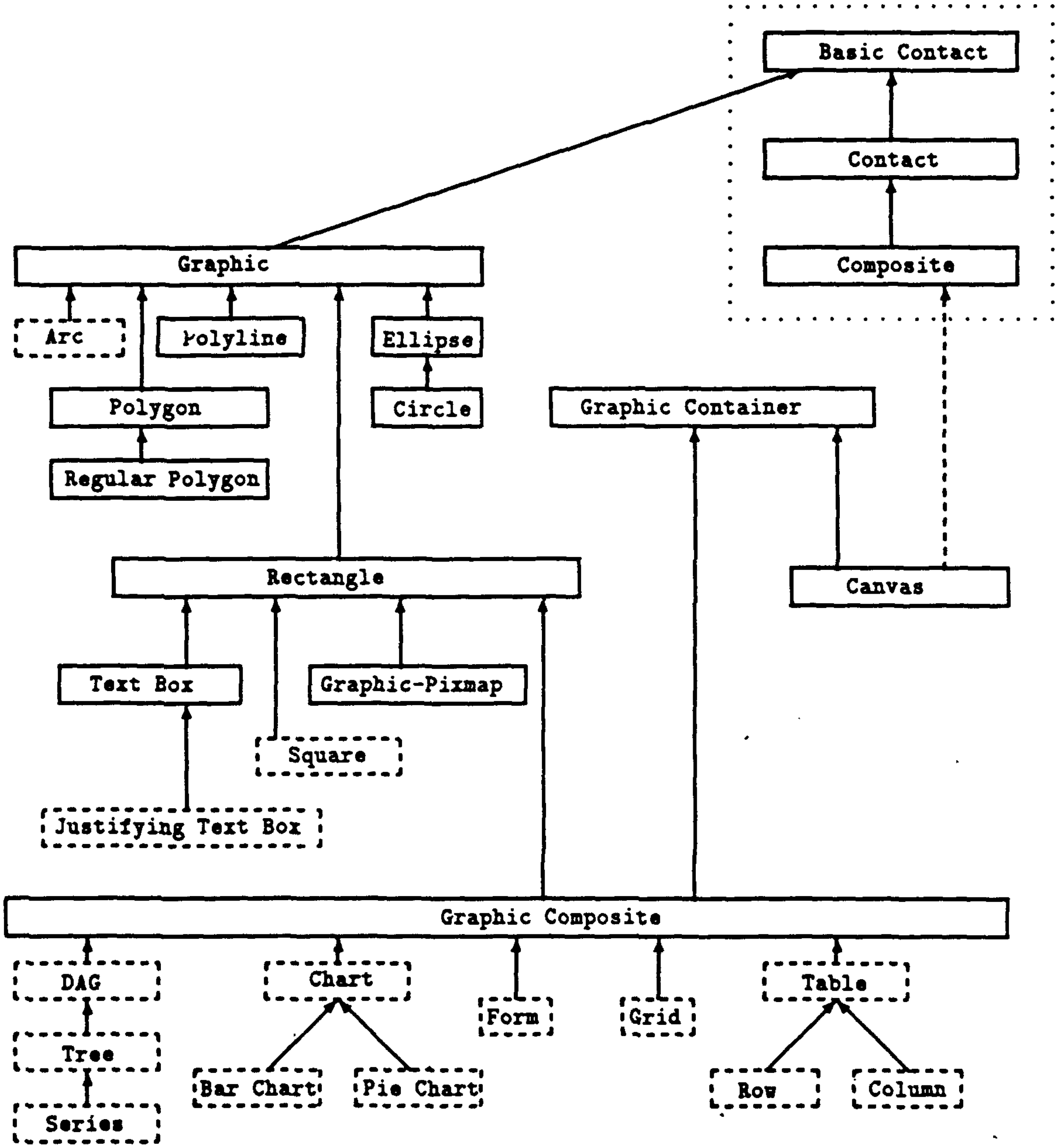

Figure 8.1: KUIE Class Hierarchy

KUIE graphics know how to organize themselves for presentation. They know how to present themselves, and how to unpresent themselves. A primitive graphic organizes itself by centering its graphical representation in a region, the dimensions of which are such that it is the smallest rectangular region that can fully encompass the graphical element (*i.e.*, its *bounding box*). Graphic-composites organize themselves by first calling down to all of their constituent parts to organize themselves. Each organization of a component part results in that component's bounding-box region (relative to its parent) being returned to the parent. The composite object then organizes these subregions into a new region. When an object is presented, it displays its graphical elements in the region defined by the object of which it is a part by calling down its list of its parts to present themselves.

The overall effect of these abilities is that the application programmer does not need to concern him/herself with *drawing* graphical images upon the display. *In fact, applications must* not *draw on a* canvas *as this will destroy the state of the window as known to KUIE.* Instead he/she simply manipulates objects — using the standard CLOS Object-oriented programming paradigms — to achieve the desired visual effects and user interactions. KUIE performs all the necessary interactions with the windowing system software to insure that the appearance of the display matches the state of the defined graphic objects and that graphic objects inform the application program of user input in an appropriate manner, when necessary.

The power of expression and versatility of CLOS makes this approach a powerful design and development tool for the construction of complex user interfaces.

8.2 Primitive Classes

The so-called "primitive" KUIE classes are those which may not have component parts; they thus form the most basic graphic elements of an application. The most basic KUIE class is the graphic class.

KUIE graphics are a subclass of the CLUE-defined class basic-contact. KUIE graphics are two-dimensional pictorial representations which are displayed on the user's screen. All communication between the user and the application is intended to be handled via graphics. The graphic class is not intended to be instantiated directly, as vanilla graphics have no defined pictorial representation. Instead, objects should be instantiated as one of the subclasses of graphic.

The graphic class defines the most primitive graphical object. All graphics have an associated region which describes the smallest rectangular region which completely encloses it (known as its *bounding box*), a canvas upon which it is drawn, a state (which is one of :withdrawn, :managed, or :mapped), and other attributes which describe its window-like characteristics (*e.g.*, background color and style, *etc.*).

8.3 Composite Classes

The an instance of one of the KUIE *primitive* classes represents a single (relatively) simple graphical object. KUIE provides for the aggregation of multiple graphic objects into a composite entity via the graphic-composite and canvas classes. These two classes are very similar but they have one important difference: graphic-composites are a subclass of KUIE graphics; canvases are a subclass of CLUE contacts. Thus, canvases are true *X Windows*; graphic-composites are not.

Nevertheless, both graphic-composites and canvases may have subsidiary objects ("children") which are subclasses of the KUIE graphic class. There are a considerable number of protocols which must be adhered to in order to properly manage graphic children. This functionality is encapsulated within the KUIE-defined graphic-container class, of which both graphic-composites and canvases are subclasses.

The graphic-container class is a mix-in class not meant to be instantiated or specialized by KUIE application programmers; KUIE graphic programmers must be aware of protocols it defines/implements, but normally will not need to deal with this class directly. For most new composite graphics, it is expected that the graphic-composite and/or canvas classes will provide a sufficient basis for complex composite graphic specializations such as tables, trees, DAGS, *etc.*

It is our intention that graphic-containers should also be capable of managing children which are CLUE contacts (which includes KUIE canvases). At the time of this writing, this capability is not fully implemented. There is nothing preventing the implementation of such a capability, however, other than a lack of time.

Graphic-composites are a subclass of rectangle and also of graphic-container — from which they inherit the ability to manage graphic children. (At some future time, graphic-composite may be promoted to be a subclass of regular-polygon, or possibly of graphic.) Although they are graphics, it is occasionally convenient to talk about them separately from the "primitive" graphic classes.

Graphic-composites define a class of graphics which provide for the hierarchical composition of graphic objects into arbitrarily complex visual arrangements. Structurally, the hierarchical composition is restricted to a strict tree (i.e., any one graphic has *exactly* one "parent").

Unlike KUIE graphics, canvases are specialized *X Windows*: they are true *X Windows* of which the *X Server* is aware. Canvases are *not* graphics, although they conform to many of the protocols defined for KUIE graphics. Cavases are a subclass of the KUIE mix-in class graphic-container and are therefore capable of of managing children which are KUIE graphics.

Since all *X Window System* calls must be directed to an *X Window*, all manipulations of KUIE graphics must eventually resolve to operations on the graphic's associated canvas. Remember that the parent of a graphic can be either another graphic (i.e., a graphic-composite) or a canvas. From any graphic object, it must be possible to reach a canvas by following the chain of parents.

When defining a new composite subclass, the KUIE graphic programmer must decide whether it should become a subclass of graphic-composite or canvas. Normally, graphic-composite is the proper choice, but there are several situations in which canvas may be more appropriate, for example:

- if the new composite will have children which are subclasses of the CLUE contact class (which includes the KUIE canvases) or must handle capabilities of *X Windows* which are presently unimplemented (*e.g.*, keyboard focus), then it must be a canvas;[4]

- in some rare circumstances it may be necessary to use canvases to achieve the desired level of performance, but this is not likely to be a consideration for most users.

Since every graphic *must* have a canvas in its chain of ancestors, it follows that every KUIE application will use at least one canvas. For *very* simple applications, it may be sufficient to instantiate an object of the class canvas directly; however most applications will create one or more canvases which have a parent that is an instance of one of the CLUE shell classes. The CLUE shell subclasses provide the necessary mechanisms for dealing with window managers, *etc.*, according to the *X Window System* protocols.

8.4 Mix-In Classes

This section describes the KUIE mix-in classes visible to an application programmer. These classes provide optional capabilities which may be mixed into application-defined subclasses of graphic as desired.

[4] As mentioned previously, these are current implementation restrictions of KUIE that eventually will be removed.

The KUIE classes connectable and connector implement a limited constraint maintenance system that permits a graphic to be notified when the geometry of another graphic changes. A common usage of connectors is in the situation where you have a line between two boxes, and you want the end-points of the line to move when either of the boxes moves (in which case you would use a subclass of connector: polyline-connector).

We anticipate that connectors and connectables will be supplanted by a more general capability when KUIE Level 2 becomes fast and robust enough. In the interim, they encapsulate a common user interface notion, namely that of "linking" a set of objects together and providing notification when any one changes its position or size.

A connector "links" instances of class connectable together. Whenever a change to the geometry of one of the connectables is made, the connector that connects them and the other connectable are notified (via **connectable-moved**). Instances of connector implement a *limited constraint mechanism*. *We anticipate that in a future release* the connectable and connector mechanism will be changed/extended to handle more general constraints as described in chapter 9.

Linking two connectables together is accomplished by creating an instance of connector, and specifying the connectables as the source and destination of the connector.

Note that instances of connector do not have a graphical representation, and can be used to "link" together connectables which do not share a common **contact-parent**. However, if both connector and a subclass of clue:basic-contact are used as superclasses of an application defined class, instances of that application defined class must have the same **contact-parent** as the **connector-destination** and **connector-source**.

A connectable provides the capability for instances of subclasses of clue:basic-contact to be "linked" together, that is to be notified when the geometry of another connectable changes. Typically connectable is used as a superclass of an application-defined graphic (or clue:contact) class so that instances of that application-defined class may be "linked" together via a connector.

Polyline-connector is a connector that has a graphical representation of a polyline. The first point in the **polyline-points** is continually updated to be the same as the **connection-point** of the **connector-source**, and similarly for the last point and the **connector-destination**. If either or both the **connector-source** or **connector-destinati^n** is nil, then the corresponding (first or last) point is not modified.

For example, polyline-connector is a convenient mechanism for representing the arcs in a graph or tree. Whenever the nodes in the graph are moved, the links are automatically updated to point to the new location of the nodes.

Chapter 9

Level 2 — Automated Layout

At the present time, KUIE Level 2 is still experimental. This section describes the purpose and operation of the KUIE automated layout system, but there is neither a reference manual or a programmer's guide available at this time.

The KUIE Layout system is a programmatic ("software" as opposed to "user") interface to KUIE geometrical object layout definition. The scope of the addressed layout problem includes object size, positioning, and relative placement. KUIE Level 2 — Layout — is a numerical interval-based constraint specification and satisfaction system that allows upper and lower quantitative bounds to be placed on well-defined object and inter-object geometric properties, *e.g.*, the width of this rectangle should be between 20 and 60 pixels ("`[20 60]`").[1]

Using an interval-based constraint system gives KUIE greater expressive power than a more conventional "flat" quantitative constraint for user interface design. To our knowledge, no other current graphical interface design environment employs an interval system. This seems to be because of its greater computational complexity (it is polynomial). However, we believe there is a body of user interface applications of significant size that will benefit from the flexibility of this technology, using the current generation of hardware workstation technology and a "bounds maintenance system" (BMS) to support fast incremental change.

[1] Interval-based constraint propagation is discussed in [25].

9.1 Motivation, Comparison to Other Work, and Goals

KUIE has been designed with the intention of relieving the programmer from the low-level "bookkeeping" work that has been common to user interface programming. KUIE Level 1 — Building Blocks — attempts to keep all but the most salient details from the programmer's purview. In Layout, our goal has been to continue in this paradigm of shifting the user interface detail-management burden from the programmer to the system.

Constraint-based graphical layout systems have been explored for almost 30 years [26] and there are presently other working constraint-based graphical user interface toolkits, including ThingLab II [27], Coral [28], Constraint Window System [29], Graphical Object Workbench [30], and Constrained Rectangular Tiled Layout [31]. Some of these toolkits also contain provision for "non-layout" (non-geometric) graphical object constraints, such active values relating graphics and other application objects or procedures.[2]

With respect to actual "layout" (geometric) constraints, all of these systems are either relatively limited in the scope of their capabilities, dealing only with a specific subset of the total layout problem (such as window tiling), or requiring that the numeric constraints specified must be exact. In KUIE, we decided that this restriction of exact numeric values may put too much of a burden on an application programmer, and we wanted to design a more flexible facility.

Thus, for example, we wanted the programmer to be able to say, "rectangle-1 is above rectangle-2," without necessarily having to say "how far" above, and have the system do something "reasonable." The interval-based constraint satisfaction approach affords the programmer the ability to leave such constraint specifications "loose." The programmer may also have notions about the minima and maxima for specific geometric quantities; the interval approach also accommodates these specifications, and uses constraint propagation to deduce their widest possible bounds across a complete, specified geometrical layout. The propagation process thus takes the place of potentially tedious and time-consuming iterative manual placement experimentation.[3]

Our overall architecture plan is broad enough to encompass the maintenance of con-

[2] These "non-geometric" constraints can be employed in KUIE applications by appropriate use of the Level 1 "connections" facilities.

[3] Presently we are working only with a programmatic specification interface. Conceivably, this could be extended into an interactive constraint specification interface; then the developer's visual sense could be more advantageously exploited, and an even better set of initial constraints for satisfaction obtained. We would like to explore this in the future.

straints during user interaction. The performance of the constraint bounds maintenance system remains in question. We believe that if we can approximate acceptable performance in this regard — even to within one or two orders of magnitude — we will have made a valuable contribution to user interface development technology. Even if it is still too early to anticipate further accelerated-performance workstations that utilize the constraint system interactively, the contributions of KUIE Layout in the programmatic regard are nonetheless unique, and powerful.

Layout is integrated with the rest of KUIE through mix-in classes for use with KUIE graphics and through hooks to the CLUE geometry manager (upon realization). A programmer may choose to use or not use the facilities of Level 2 Layout for various parts of the application user interface. (That is, exact numeric bounds may still be used for any geometric quantities, and no objects need necessarily be constrained.)

9.2 Capabilities and Architecture

Layout consists of three main existing parts, and one planned part:

1. the **Specifier**, including graphical object and constraint definitions;
2. the **Assimilator**, an arithmetic propagation engine that refines all known bounds to their least justified quantitative span and detects any bounds or constraint inconsistencies, producing a consistent, completely propagated assimilation;
3. the **Allocator**, which performs freespace allocation decisions that fix underconstrained (still greater-than-zero-length-interval) bounds to an exact numeric value from the allowed set which lead to harmonious object relative placement within total (previously uncommitted) freespace; and,
4. the (planned) BMS ("bounds maintenance system"), a special purpose, streamlined reason maintenance system to support enhanced interactive operation.

These parts are invoked under program control.

9.2.1 The Specifier

For the sake of generality, Layout deals specifically with object rectangle (*i.e.*, *bounding-box*) definitions; since all KUIE graphic objects have an associated *bounding-*

box, Layout treats all graphics as if they were rectangles. A KUIE rectangle specification has the following form:

```
(setf r1 (make-instance 'rectangle
   :height [20 30] :width [40 50] :top [0 100]
   :bottom [0 400] :left [20 600] :right [80 800]))
```

Any initialization field left unspecified is "unconstrained;" that is, it takes an interval value of `[0 :infinity]`.

Some examples of higher-level Layout specifications are shown below:

```
(place r1 :left-of r2 :offset [100 200])
(place r1 :left-of r2 :overlapping [0 25])
(align (top r1) :to (top r3))
(align (y (center r7)) :to (y (center r1)))
(scale (width r7) :to (width r1) :by 2)
(equate (aspect-ratio r3) :with 1)
(scale (area r3) :to (area r2) :by [3/2 5/2])
```

Specifically omitted from Layout "placement" specifications are notions of negation and disjunction, *i.e.*, "r1 is not above r2," or "r2 is left-of or right-of r1." Including these would, in general, make the layout problem intractable. This is a compromise: our constraint satisfaction machinery is already polynomial-complexity; adding disjunction would make it exponential. For example, one commonly desired specification capability would be that "r1 is near-to r2 without overlapping." In our system this would have to map into the four-way disjunction: "r1 is either left-of r2 or right-of r2 or above r2 or below r2," with appropriate offset distance specifications in each direction. We have considered the possibility of including such "conditional constraints," and we would like to experiment with structured ways of using them. A utility to help a user understand the computational size of his problem and the origin of its complexity would also be useful.

Other planned layout specification capabilities include proportioned rows/columns (*i.e., tiling*), nested composite rectangle coverage, and row or column *wrapping* of child objects into an arrayed representation.

In addition to the "standard" types of constraint specifications represented above, we also allow the specification of "arbitrary" arithmetic constraints over geometric quantities, using the operations of addition, subtraction, multiplication, division, square, and square-root. Indiscriminate use of arbitrary constraint relations, may, however, lead to trouble — see below.

9.2.2 The Assimilator

The Assimilator is a simple Waltz-style propagator, as descnbed in [25]. Geometric quantities, such as height, area, x-coordinate, or a scale factor, are represented as *nodes* with interval bounds. *Constraints* relate these nodes arithmetically. When constraints are initially asserted or whenever a node which is a member of a constraint is "changed," the constraint is *queued* for refinement processing. In refinement, each constraint node is evaluated arithmetically with respect to the other nodes, and if the evaluation result is "tighter" — with either a higher lower-bound or a lower upper-bound than the node under refinement, then that node is changed to reflect the tighter bounds warranted by the current state of the system. The algorithm terminates when the entire network represented by such constraints becomes quiescent.

Arithmetic interval constraint propagation systems have some cantankerous properties, most of which we have been able to avoid in this relatively restricted problem domain. Under some "pathological" initial conditions, they may not terminate. So far, the "sensible" constraint specifications that we have developed are oriented to intuitive and apparently well-behaved properties of Euclidean geometry, and have not been pathological. Under poor constraint selection ordering, execution time may become exponential. We employ a "stratified" system of separate queues based on different arithmetic constraint types, in order to maintain some control over constraint refinement order. Complexity analysis for interval propagation systems is difficult and depends on the kinds of arithmetic constraint operations employed. So far, we have not done a formal complexity analysis, but our initial experiments show that, at least fcr some problems, execution time grows at less than the square of the problem size in number of nodes.

The input of assimilation is a set of "unrefined" nodes; its output is a set of nodes with intervals that are consistent with all of the constraints of the system. The Assimilator has some limitations on computational power that also demand extra care when writing constraint expressions.

In particular, the Assimilator can perform no equality substitution (or "term-rewriting"), either across constraint arithmetic expressions or within single constraints — even though there clearly are computational situations where term, or "node," substitutions are necessary to solve for the best bounds on nodes. We made this decision since general term-rewriting mechanisms are intractable for propagation. The danger in this is that a value picked for a node whose assimilated bounds are not the best logically warranted bounds might not actually be logically consistent with the rest of the system. This would become a problem in allocation. A way out of this computational impoverishment is to assert additional constraints that solve equality relationships for you, in effect, performing the required substitutions in them, "by hand." So far, this has been satisfactory: the problem has come up only in constraints employing square and square-root operations, and we have coded it into our "primitives" for rectangle area and point-to-point straight-line distance. A section of the (eventual) Layout users manual will flag this problem potential, and, at any rate, it only applies to "custom-generated" arbitrary constraints.

9.2.3 The Allocator

One remaining challenge to using the interval-based approach has been to selecting particular values for each node in the system. The Assimilator only refines interval bounds to the set of what is mutually consistent among all constraints. This, as Davis points out ([25]), in effect only defines the "Cartesian product" of the solution space over all nodes; while it is true (given a set of constraints with adequate hand-substituted "rewrites" to make it logically complete) that for any one node a value can be chosen that when taken with the rest of the system is consistent, there is no guarantee that you can do this for two different nodes at the same time. Extracting a particular solution from a fully assimilated network still requires additional work. This is what the Allocator is for.

A very general allocator — one that was always guaranteed to work — would simply loop through all system nodes, choosing an exact value (allocation) for each one, and then reassimilating. Such a process would not necessarily be the most efficient possible, or even generate a very pretty picture. Our Allocator uses information about parent-rectangle edge-to-edge child-rectangle paths and path-lengths, in the x- and y-dimensions, to determine an intuitively "centering" allocation that is also relatively efficient, in that it allocates all of the nodes on a path in one step, between assimilations.

Our approach is to work first on the path with minimum "freespace" — the difference between parent edge-to-edge length and the sum of node interval lower bounds along the path. By choosing the path with the minimum freespace, we tend to fix its length nodes in a "centered" position, before going on to other, shorter-bounded paths. Choosing these shorter paths (with less freespace) first leads to allocations in which later-allocated rectangles (rectangles whose lengths are allocated later) end up bunched up at one end of the picture. Our Allocator includes a minimum-freespace path search algorithm that also caches best paths between assimilations.

These decisions in our Allocator are relatively arbitrary, but they are also straightforward. Any alternative must somehow prioritize nodes for allocation order and also provide formulas, or "allocation directives" that pin down values from intervals.

9.2.4 The BMS — Bounds Maintenance System

The BMS is a (planned) special-purpose, streamlined reason maintenance system to support fast incremental change and enhanced interactive operation. Interval bounds systems respond well to addition of new constraints but include no provision for their deletion or retraction without complete network reassimilation. We expect the BMS to support incremental deletion in assimilation and incremental deletion and addition in allocation. It will work by recording constraint-and-node dependencies for all node refinements and allocation decisions. The BMS concept is also essential in the (possible) implementation of conditional constraints, since it will supply the machinery for "backtracking" through propagation, and also as the basis for more robust constraint violation exception handling, for the same reason.

Our initial experiments with the BMS concept show that it may be a feasible approach. While we have not yet implemented even a prototype BMS system, our reasoning is as follows. It is, typically, much less expensive to add a new constraint to an already assimilated system and then reassimilate than it is to reassimilate the whole system over again from scratch. By analogy, the "rollback" of a deleted constraint should also be, typically, of a much smaller order than the reassimilation process too. Taken together (in sequence), these two processes — deletion/rollback plus incremental reassimilation — should still be significantly less expensive than wholesale reassimilation. We expect to have the prospect of BMS feasibility confirmed, through implementation, one way or the other in the near future.

While the performance of the BMS remains in question, we believe that if we can approximate acceptable performance in this regard — even to within one or two orders of magnitude — we will have made a valuable contribution to user interface development technology. Even if it is still too early to anticipate further accelerated-performance workstations that utilize the constraint system interactively, the contributions of KUIE Layout in the programmatic regard are unique, and powerful.

9.3 Example

Here is an example KUIE Layout specification:

```
(let ((wd [40 50])
      (ht [20 30])
      (parent (make-rectangle :width 800 :height 600)))
   (make-rectangle 'r1 :width wd :height ht :parent parent)
   (make-rectangle 'r2 :width wd :height ht :parent parent)
```

```
  (make-rectangle 'r3 :width wd :height ht :parent parent)
  (make-rectangle 'r4 :height ht :parent parent)
  (make-rectangle 'r5 :parent parent)
  (make-rectangle 'r6 :parent parent)
  (make-rectangle 'r7 :parent parent)
  (place r1 :above r2) (align (x (center r1)) :to (x (center r2)))
  (place r3 :left-of r1 :overlapping t)
  (place r4 :above r1) (align (center r4) :to (left r1))
  (scale (width r4) :to wd :by 3)
  (place r5 :right-of r1) (align (y (center r5)) :to (y (center r1)))
  (scale (width r5) :to (width r1) :by [1 2])
  (scale (height r5) :to (height r1) :by [2 3])
  (place r6 :right-of r5) (align (y (center r6)) :to (y (center r5)))
  (equate (aspect-ratio r6) :with [1 2])
  (scale (area r6) :to (area r5) :by [2 3])
  (scale (left r7) :to (left r5) :by [3 5]))
```

It creates seven child rectangles. The first three share common width nodes and the first four share common height nodes, from the lexical variables wd and ht. Three *heights, four widths, and all coordinates* are unconstrained. Most of the specifications are straightforward; in the second scaling constraint, (width r5) is scaled to (be, in this case, bigger than) (width r1) by a factor of from [1 2]. The "equating" specification identifies r6's aspect ratio with an interval. Immediately after specification, the seven rectangles have nodes with the following bounds:

```
(("R1" :WIDTH [40 50] :LEFT [0 :INFINITY] :RIGHT [0 :INFINITY]
       :HEIGHT [20 30] :TOP [0 :INFINITY] :BOTTOM [0 :INFINITY]
       :HANDLES ((CENTER :X [0 :INFINITY] :Y [0 :INFINITY])))
 ("R2" :WIDTH [40 50] :LEFT [0 :INFINITY] :RIGHT [0 :INFINITY]
       :HEIGHT [20 30] :TOP [0 :INFINITY] :BOTTOM [0 :INFINITY]
       :HANDLES ((CENTER :X [0 :INFINITY] :Y [0 :INFINITY])))
 ("R3" :WIDTH [40 50] :LEFT [0 :INFINITY] :RIGHT [0 :INFINITY]
       :HEIGHT [20 30] :TOP [0 :INFINITY] :BOTTOM [0 :INFINITY])
 ("R4" :WIDTH [1 :INFINITY] :LEFT [0 :INFINITY] :RIGHT [0 :INFINITY]
       :HEIGHT [20 30] :TOP [0 :INFINITY] :BOTTOM [0 :INFINITY]
       :HANDLES ((CENTER :X [0 :INFINITY] :Y [0 :INFINITY])))
 ("R5" :WIDTH [1 :INFINITY] :LEFT [0 :INFINITY] :RIGHT [0 :INFINITY]
       :HEIGHT [1 :INFINITY] :TOP [0 :INFINITY] :BOTTOM [0 :INFINITY]
       :HANDLES ((CENTER :X [0 :INFINITY] :Y [0 :INFINITY]))
       :PARAMETERS ((ASPECT-RATIO [:-INFINITY :INFINITY])
          (AREA [:-INFINITY :INFINITY])))
 ("R6" :WIDTH [1 :INFINITY] :LEFT [0 :INFINITY] :RIGHT [0 :INFINITY]
       :HEIGHT [1 :INFINITY] :TOP [0 :INFINITY] :BOTTOM [0 :INFINITY]
       :HANDLES ((CENTER :X [0 :INFINITY] :Y [0 :INFINITY]))
       :PARAMETERS ((AREA [:-INFINITY :INFINITY])
          (ASPECT-RATIO [:-INFINITY :INFINITY])))
 ("R7" :WIDTH [1 :INFINITY] :LEFT [0 :INFINITY] :RIGHT [0 :INFINITY]
       :HEIGHT [1 :INFINITY] :TOP [0 :INFINITY] :BOTTOM [0 :INFINITY]))
```

Except for the width and height nodes we set explicitly, all nodes are unconstrained, except for system-defined limits. After assimilation, the rectangles' nodes have the following bounds:

```
(("R1" :WIDTH [40 50] :LEFT [60 226] :RIGHT [100 266]
       :HEIGHT [20 30] :TOP [20 560] :BOTTOM [40 580]
       :HANDLES ((CENTER :X [80 246] :Y [30 570])))
 ("R2" :WIDTH [40 50] :LEFT [55 226] :RIGHT [100 271]
       :HEIGHT [20 30] :TOP [40 580] :BOTTOM [60 600]
       :HANDLES ((CENTER :X [80 246] :Y [50 590])))
 ("R3" :WIDTH [40 50] :LEFT [10 226] :RIGHT [60 266]
       :HEIGHT [20 30] :TOP [0 580] :BOTTOM [20 600])
 ("R4" :WIDTH [120 150] :LEFT [0 166] :RIGHT [120 301]
       :HEIGHT [20 30] :TOP [0 540] :BOTTOM [20 560]
       :HANDLES ((CENTER :X [60 226] :Y [10 550])))
 ("R5" :WIDTH [40 100] :LEFT [100 266] :RIGHT [140 366]
       :HEIGHT [40 90] :TOP [0 550] :BOTTOM [50 600]
       :HANDLES ((CENTER :X [120 316] :Y [30 570]))
       :PARAMETERS ((ASPECT-RATIO [0 2]) (AREA [1600 9000])))
 ("R6" :WIDTH [40 329] :LEFT [140 760] :RIGHT [180 800]
       :HEIGHT [40 164] :TOP [0 550] :BOTTOM [50 600]
       :HANDLES ((CENTER :X [160 780] :Y [30 570]))
       :PARAMETERS ((AREA [3200 27000]) (ASPECT-RATIO [1 2])))
 ("R7" :WIDTH [1 500] :LEFT [300 799] :RIGHT [301 800]
       :HEIGHT [1 600] :TOP [0 599] :BOTTOM [1 600]))
```

These are the widest bounds that are consistent with all of the input specifications. Note that all of the child rectangles' coordinates fit within the dimensions of the parent rectangle. Now the bounds must be allocated. This is the only way to get a firm, meaningful picture to display. After allocation, the nodes have the following values:

```
(("R1":WIDTH 44 :LEFT 144 :RIGHT 189 :HEIGHT 28 :TOP 285 :BOTTOM 314
      :HANDLES ((CENTER :X 167 :Y 300)))
 ("R2" :WIDTH 44 :LEFT 144 :RIGHT 189 :HEIGHT 28 :TOP 442 :BOTTOM 471
      :HANDLES ((CENTER :X 167 :Y 457)))
 ("R3" :WIDTH 44 :LEFT 121 :RIGHT 166 :HEIGHT 28 :TOP 285 :BOTTOM 314)
 ("R4" :WIDTH 134 :LEFT 77 :RIGHT 211
      :HEIGHT 28 :TOP 128 :BOTTOM 157
      :HANDLES ((CENTER :X 144 :Y 143)))
 ("R5" :WIDTH 79 :LEFT 215 :RIGHT 295 :HEIGHT 82 :TOP 258 :BOTTOM 341
      :HANDLES ((CENTER :X 255 :Y 300))
      :PARAMETERS ((ASPECT-RATIO 1) (AREA 6528)))
 ("R6" :WIDTH 181 :LEFT 470 :RIGHT 652 :HEIGHT 104 :TOP 247 :BOTTOM 352
      :HANDLES ((CENTER :X 562 :Y 300))
      :PARAMETERS ((AREA 18953) (ASPECT-RATIO 2)))
 ("R7" :WIDTH 61 :LEFT 677 :RIGHT 739 :HEIGHT 200 :TOP 199 :BOTTOM 400))
```

The allocated configuration is shown in figure 9.1. This example makes rather liberal use of interval-valued and unconstrained nodes, with only one scale factor pinned down to an exact value before assimilating. In general, the more allocated values there are in a system specification, the faster it will go; this efficiency differential also applies to dependency management and incremental change.

9.4 Status and Completion Plans

The specifier, assimilator, and allocator, are all existing and working in initial implementations. We can get nice pretty pictures on the screen. Execution times for the sum of the three operations appear to be on the order of 1 second per object, for fairly large (50-rectangle) constraint systems, running on the SUN4 Sparcstation.

Our planned development activities for Layout include the following:

Extension to composite objects. (The first pass was for a single parent system.) Our plan is to provide a unitary (multi-)parent-child constraint satisfaction system.

Incremental object/constraint deletion support. Currently, because we are running with an interval bounds system, there is no support for deletion without completely reassimilating; we are looking into the design of a special-purpose, streamlined constraint bounds TMS (truth maintenance system) to support incremental deletion and enhanced interactive operation. Working with interval

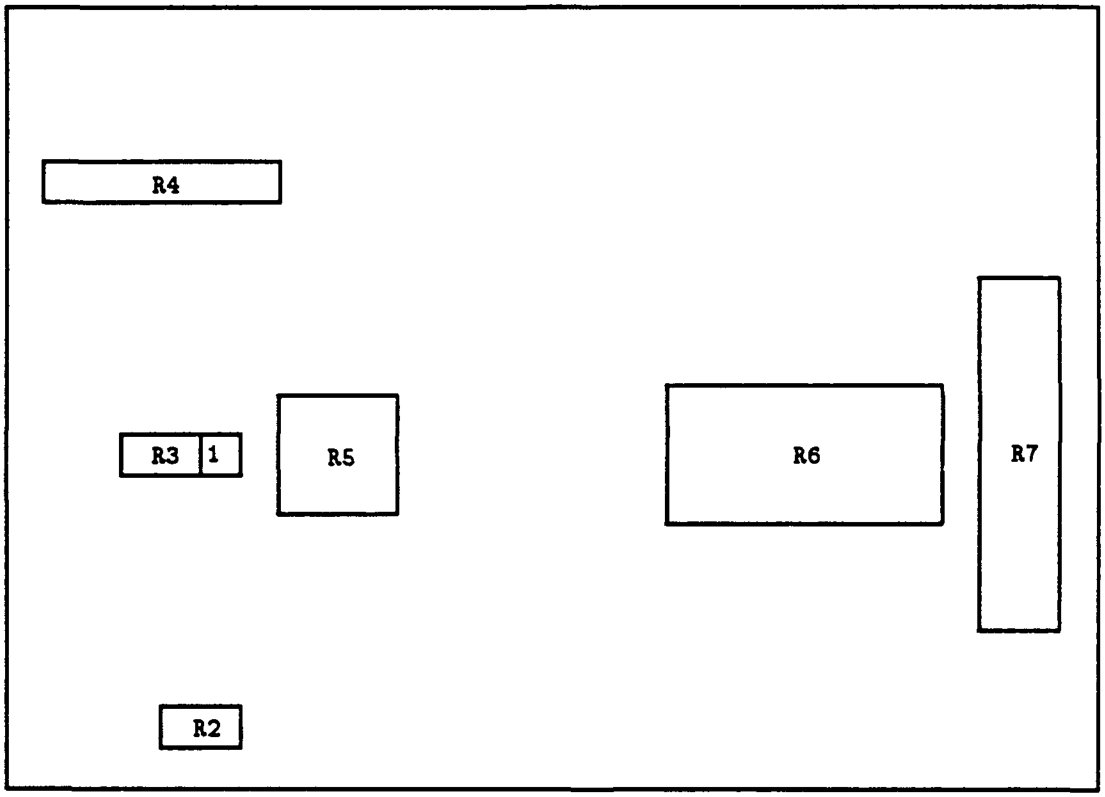

Figure 9.1: KUIE Level 2 Example

bounds, however, gives us much more freedom and power than constraint-based user interface design systems requiring ful!y-constrained specification.

Tiling and wrapping operations. As described above.

Investigation of KUIE Levels 2 and 3 interactions and integration. This is a low priority, design-only item.

Layout documentation. Something like a user's manual would be nice.

Bibliography

[1] Green, C., Luckham, D., Balzer, R., Cheatham, I. and Rich, C., "Report on a Knowledge-Based Software Assistant, " prepared for Rome Air Development Center, Griffiss AFB, New York 13441, June 15, 1983.

[2] KBSA Framework Final Technical Report Phase 1, Honeywell Systems and Research Center, Minneapolis MN, October, 1988.

[3] "The Wisdom Systems Concept Modeler Reference Manual", McDermott International, 1989.

[4] Carciofini, J., Colburn, .T, Hadden, G., Larson, A., "LogLisp Programming System Users Manual", Honeywell Systems and Research Center, July 23, 1987.

[5] "Department of Defense Trusted Computer System Evaluation Criteria" CSC-STD-001-83, Department of Defense, Computer Security Center, Fort George G. Meade, Maryland, August, 1983.

[6] Scheifler, R., and Newman, R., "X Window System Protocol, Version 11," Massachusetts Institute of Technology, Cambridge, Massachusetts and Digitial Equipment Corporatior Maynard, Massachusetts, 1987.

[7] Clark, J., Larson, A., and Schrag, B., KBSA User Interface Environment (KUIE) Reference Manual, Honeywell Systems and Research Center, Minneapolis MN, June 1990.

[8] Kimbrough, K. and Oren, L., "Common Lisp User Interface Environment (CLUE)," Texas Instruments, July 1989.

[9] Myers, B.A., et al., "The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp", Carnegie Mellon Univerity, Computer Sciences Department, CMU-CS-90-117, March 1990.

[10] Tichy, W., "RCS – A System for Version Control," *Software-Practice and Experience*, Vol. 15(7), July 1985.

[11] Perry, D., "Version Control in the Inscape Environment," *9th International Conference on Software Engineering*, IEEE, March 30, April 2, 1987.

[12] Katz, R., *Information Management for Engineering Design*, Springer-Verlag, 1985.

[13] EIS Specification Volume I: Organization and Concepts, (CDRL 13,16,17 under WRDC contract F33615-87-C-1401), October 1989.

[14] Rumbaugh, J., "Controlling Propagation of Operations using Attributes on Relations," *OOPSLA '88 Proceedings*, September 25-30, 1988.

[15] Katz, R., "Towards a Unified Framework for Version Modeling," University of California, Berkeley.

[16] Vines, P., Vines, D., King, T., "Configuration and Change Control in Gaia," Honeywell Systems and Research Center, Minneapolis MN.

[17] Steel, G.L., *Common Lisp — The Language*, Digital Press, 1984.

[18] Bobrow, D.G., *et al*, "Common Lisp Object System Specification," *Draft submitted to ANSI X3J13, June 1988, 88-002R.*

[19] "Knowledge Based Requirements Assistant," Interim Technical Report, Sanders Associates, March 17, 1986.

[20] "Common LISP X Interface (CLX)," Texas Instruments, 1989.

[21] Kimbrough, K., et al, "Common Lisp Interactive Objects (CLIO)," Texas Instruments, September 1989.

[22] "Programming The User Interface", Symbolics Lisp Machine Documentation set, Chapter 7.

[23] Krasner, G. and Pope, S., "A Cookbook for using the Model-view-Controller User Interface Paradigm in Smalltalk-80," ParcPlace Systems, 1988.

[24] London, R. and Duisberg, R., "Animating Programs Using Smalltalk," IEEE Computer, August 1985.

[25] Davis, E., "Constraint Propagation through Interval Labels," *Artificial Intelligence* **32:3**, July 1987, 281–331.

[26] Sutherland, I., "Sketchpad: A Man-machine Graphical Communication System," *Proceedings of the Spring Joint Computer Conference* (IFIPS), 1963, 329–345.

[27] Maloney, J.H., Borning, A., and Freeman-Benson, B.N., "Constraint Technology for User-Interface Construction in ThingLab II," *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages, and Applications* (OOPSLA), 1989, 381–388.

[28] Szekely, P.A. and Myers, B.A., "A User Interface Toolkit Based on Graphical Objects and Constraints," *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages, and Applications* (OOPSLA), 1988, 36–45.

[29] Epstein, D. and LeLonde, W.R., "A Smalltalk Window System Based on Constraints," *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages, and Applications* (OOPSLA), 1988, 83–94.

[30] Barth, P.S., "An Object-oriented Approach to Graphical Interfaces," *ACM Transaction on Graphics,* 5:2, April 1986, 142–172.

[31] Cohen, E.S., Smith, E.T., and Iverson, L.A., "Constraint-based Tiled Windows," *IEEE Computer Graphics and Applications,* May 1986, 35–45.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.